GPGPUによる頻出部分グラフマイニングの実装とその評価

Implementation and Evaluation of GPU-based Frequent Subgraph Miner

土岐 達哉[1]　尾崎 知伸[1*]
Tatsuya Toki[1], Tomonobu Ozaki[1]

[1] 日本大学大学院 総合基礎科学研究科
[1] Graduate School of Integrated Basic Sciences, Nihon University

Abstract: Frequent subgraph mining, *i.e.* enumeration of all subgraph patterns appearing frequently in a given database, is one of the fundamental problems in graph mining. In general, subgraph miners need to solve the subgraph isomorphism. Thus, they require long computation time, if target databases are huge or given frequency threshold is very low. To overcome this computation complexity, several optimization techniques as well as parallel implementations have been developed. In this research, we investigate on the applicability of GPGPU to subgraph mining. We re-implement a recent developed GPU-based subgraph miner[5] and optimize the miner through profiling. As a result, we achieve about two times speed-up. In addition, to obtain certain insights on the performance characteristics of the miner, we conduct systematic evaluation by using carefully prepared synthetic datasets.

1　はじめに

頻出部分グラフマイニングとは，複数のグラフで構成されるデータベースから頻出するパターン，すなわち部分グラフを列挙するマイニング手法である．以前から化合物や文書などをはじめとする多くの実データがグラフで表現され，グラフマイニングの対象とされてきた．近年ではソーシャルネットワークを対象としたビッグデータが注目を集め，それに伴い，再びグラフマイニングの需要が高まりつつある．

これまでに幅優先探索に基づく AGM[2] や FSG[3]，深さ優先探索に基づく gSpan[1] や Gaston[4] など，頻出部分グラフを列挙するアルゴリズムが数多く提案されている．また計算時間の短縮を目的とした並列アルゴリズムも数多く提案されている [6, 7, 8, 9, 10, 11]．特に近年では，画像処理を行う GPU を用いて汎用計算を行う GPGPU(General-purpose computing on graphics processing units) を用い，計算時間の短縮を試みる研究が提案されている [5, 12]．本論文では，文献 [5] で提案された，深さ優先探索に基づく代表的な頻出部分グラフ列挙アルゴリズム gSpan の GPGPU を用いた実装 PGM に着目する．PGM は，主に頻度の計算と出現（埋め込み）の拡張に GPU を用いており，いくつかの実データに対して良好な並列性を有していることが報告されている [5]．しかし，多様なデータを用いた系統的・総合的な評価が行われているわけではなく，データの特質が実行速度に与える影響など実利用におけるシステムの性質などは必ずしも明らかではない．

そこで本論文では，NVIDIA 社が提供する CUDA を用いて PGM を再実装するとともに，プロファイル技術を用いた実装レベルでの改善を行ったうえで，様々なパラメタを調整した系統的な人工データを用い，その性能を多角的に評価する．

本論文の構成は以下のとおりである．2 章で頻出部分グラフマイニングの形式的な定義を与える．3 章では文献 [5] に基づき，gSpan アルゴリズムの GPU による実装について概観する．4 章では，CUDA を用いて実装したプロトタイプとプロファイルングに基づく改善について述べる．5 章で実験と考察を行い，最後に 6 章でまとめと今後の課題を述べる．

*連絡先： 日本大学大学院 総合基礎科学研究科
東京都世田谷区桜上水 3-25-40
E-mail:ozaki.tomonobu@nihon-u.ac.jp

2　頻出部分グラフマイニング

本研究では，グラフを多重辺や自己ループを許さないラベル付き連結無向グラフに限定する．

グラフ $G = (V, E, L)$ を，頂点集合 $V = \{v_1, \cdots, v_m\}$ とエッジ集合 $E \subseteq V \times V$，ラベル付け関数 $L : V \cup E \to \mathcal{L}$ の 3 項組とする．ここで $\mathcal{L}$ はラベルの集合である．

また n 個のグラフから構成されるグラフデータベースを $\mathcal{D} = \{G_1, \cdots, G_n\}$ と表記する．

グラフデータベース $\mathcal{D}$ に対し，パターン P の支持度 (support) を

$$support(P) = |\{G_i \mid G_i \in \mathcal{D}, P \subseteq G_i\}| \ / \ |D|$$

と定義する．なお $P \subseteq G$ とは G 中に P が出現する，すなわち P が G の誘導部分グラフであることを表す．

頻出部分グラフマイニングとは，データベース $\mathcal{D}$ と最小支持度 $minsup > 0$ が与えられた時，$support(P) \geq minsup$ を満たす全ての頻出パターンを列挙する問題である．この問題は，パターン空間を対象とした探索問題として定式化される．これまでに幅優先探索に基づくアルゴリズム AGM[2] や深さ優先探索に基づくアルゴリズム gSpan[1] などが提案されている．

本研究で対象とする gSpan は，標準形判定を伴う最右拡張を用いた逆探索に基づき，深さ優先に頻出部分グラフを列挙するアルゴリズムであり，多くの手法の基礎を与えている．gSpan では，各部分グラフパターンを DFS コードと呼ばれる文字列で表現する．部分グラフパターンの全域木（DFS 木）を考え，各辺 $e = (v, u)$ を $(v, u, L(v), L(v, u), L(u))$ の 5 つ組のタプルで表し，これらを順に並べることで，部分グラフパターンの文字列表現，すなわち DFS コードとする．グラフには様々な同型が考えられ，それらの全域木の取り方も複数考えられるが，ある基準で最小となる DFS コードを与えるグラフを標準形と考える．さらに，標準形グラフのみを列挙することで，同型グラフの重複列挙を回避している．

あるグラフ G に対する DFS 木 T が与えられたとき，DFS コードを $code(G, T)$ と記す．$G, T, \alpha = code(G, T)$ が与えられたとき，$\alpha = (a_0, a_1, \cdots, a_m)$, $m \geq 2$ を仮定する．このとき $a_k = (i_k, j_k, l_{i_k}, l_{(i_k, j_k)}, l_{j_k})(0 \leq k \leq m)$ と定義する．

アルゴリズム 1 に，(簡略化した）gSpan の列挙アルゴリズムを示す．ここで関数 RME(P) は，パターン P に最右拡張を適用することで得ることのできるパターンの集合である．また，minDFS(P) は，P の標準形を返す関数である．

アルゴリズム 1 gSpan($\mathcal{D}$, $minsup$)

```
F ≔ {}
E ≔all frequent 1-edge graphs in D
for each e ∈ E
  gSpan(e, D, minsup, F)
end
return F
```

Procedure gSpan(P, $\mathcal{D}$, $minsup$, F)

```
if P ≠ minDFS(P) then return;
if support(P)< minsup then return;
F ≔ F ∪ {P};
for each P' in RME(P)
  gSpan(P', D, minsup, F);
end
```

3 gSpan への GPU の適用 [5]

近年，Kessl らにより，GPGPU を用いた gSpan の実装が提案された [5]．本論文では，便宜上，この実装を PGM と呼ぶ．

アルゴリズム 2 に，PGM の主要部分のアルゴリズムを示す．このアルゴリズムでは 1 行目でパターン P から拡張可能な全ての辺を” 拡張” として得る．2 行目で 1 行目で得られた” 拡張” の支持度を並列に計算し，頻出でないものを削除する．3 行目以降の for 文内では，残った頻出の” 拡張” をパターン P に追加し，最小 DFS コードを満たすか確認し，パターンを出力している．また，出力した後に拡張したパターンの埋め込みを GPU で計算し，再帰処理を行っている．

このように PGM では，

1. パターン P の拡張を得る処理
2. 拡張されたパターンの支持度を計算する処理
3. パターンの埋め込み（出現位置のリスト）を得る処理

に GPU が用いられている．またこれらの処理に合わせ，メモリ量の少ない GPU 向けのデータ構造が提案されている．本章では，これらの技法について概説する．

3.1 データ構造

データベース $\mathcal{D}$ を GPU 上に展開するとき，各グラフの頂点番号をデータベース中で一意に定まるような $global\ vertex\ id$ と呼ばれる番号に変換する．データベースは頂点集合 V の近傍の $global\ vertex\ id$ を持つ一次元配列 N と各頂点の N 内での先頭要素のインデックス番号を持つオフセット配列 O で表すことが出来る．図 1 は，$\mathcal{D} = \{G_1, G_2\}$ について，$global\ vertex\ id$ と頂点番号を示したものである．(括弧内が頂点番号）

PGM と同様に図 1 を N と O で表したものを図 2 に示す．

元々の gSpan アルゴリズムでは，新たな部分グラフパターンが生成されるたびに，各データに対して部分グラフ同型の有無の計算が行われ，そのパターンの支

アルゴリズム 2 Graph Mining on GPUs (Database $\mathcal{D}$, Threshold $minsup$, Pattern P, Embeddings $\sum_{\mathcal{D}}(P)$)

(GPU step) Get all possible edge extensions $\varepsilon_{\mathcal{D}}(P)$ of P
(GPU step) Compute support for all extensions in $\varepsilon_{\mathcal{D}}(P)$ and remove infrequent extensions
for each $e = (v_i, v_j, l_i, l_{ij}, l_j) \in \varepsilon_{\mathcal{D}}(P)$ do
  $P' \leftarrow P$ extended by e
  if $P' = minDFS(P')$ then
    output P'
    (GPU step) Create $\sum_{\mathcal{D}}(P')$
    $GRAPHMINING(\mathcal{D}, minsup, P', \sum_{\mathcal{D}}(P'))$
  end if
end for

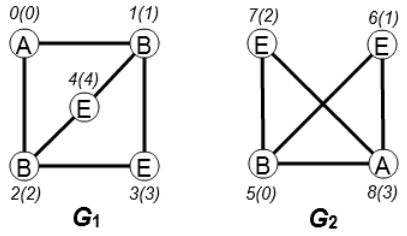

図 1: $\mathcal{D}$ の *global vertex id* (*pattern vertex id*)

global id	0		1			2			3		4	
N	1	2	0	3	4	0	3	4	1	2	1	2

global id	5			6		7		8		
$N(cont'd)$	6	7	8	5	8	5	8	5	6	7

index	0	1	2	3	4	5	6	7	8
O	0	2	5	8	10	12	15	17	19

図 2: グラフデータベースの GPU 表現
（[5] より引用）

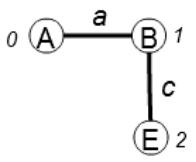

図 3: グラフ G

持度が計算される．これに対し PGM では，パターンが最右拡張によってのみ生成されることに着目し，対象となるパターンの埋め込み（パターンの出現位置）を保持するとともに，それを適宜拡張することで，新たなパターンの埋め込みと支持度を計算する．この処理には GPU が用いられるが，そのためのデータ構造を導入する．図 3 のグラフ $G = \{A, B, E\}$ を頂点数 $p = 3$ のパターン P として $\mathcal{D}$ 内での埋め込みを得たときの例を図 4 に示す．埋め込みの先頭部分を共通化した Prefix tree から (idx, vid) からなる $Q_{1,\cdots,p}$ を作ることができる．idx は前列のインデックスを表し，vid は *global vertex id* を表す．このとき Q_1 の idx を-1 とすることにより，頂点 $p \to p-1 \to \cdots \to 1$ と，idx が-1 になるまで辿ることで埋め込みを獲得でき，かつ先頭要素が共通化されていることでメモリの節約も達成される．

Q_1	
idx	vid
-1	0
-1	0
-1	8

Q_2	
idx	vid
0	1
1	2
2	5

Q_3	
idx	vid
0	3
0	4
1	3
1	4
2	6
2	7

図 4: 埋め込み列 Q（[5] より引用）

3.2 並列演算

並列プリミティブ演算とは，Prefix sum や Reduction のような演算のことであり，これらは GPU による高速化の恩恵を受けやすい演算である．アルゴリズムを GPU 向けに変換する場合，いかにしてこれらの演算が可能な形式へ落とし込むかが重要な点となる．

GPU では，多数のスレッドにより処理が並列かつ独立して実行される．ここでは，グラフの拡張方法の並列処理について解説する．グラフの拡張とは，例えば $P = (\mathcal{R}(P) =)\{A, B\}$ について探索中に各頂点から伸ばすことが可能な，つまり $\mathcal{D}$ 内に存在するエッジを発見することである．まずは埋め込み列の最右パス $\mathcal{R}(P)$ を列優先の 2 次元配列に展開し，図 5 のように各スレッドにそれぞれの頂点を割り当てる．

$0_{(0)}$	$6_{(1)}$	$12_{(3)}$
$1_{(0)}$	$7_{(1)}$	$13_{(4)}$
$2_{(0)}$	$8_{(2)}$	$14_{(3)}$
$3_{(0)}$	$9_{(2)}$	$15_{(4)}$
$4_{(8)}$	$10_{(5)}$	$16_{(6)}$
$5_{(8)}$	$11_{(5)}$	$17_{(7)}$

図 5: スレッド割り当て ($thread\text{-}id_{(vertex\text{-}id)}$)
（[5] より引用）

各スレッドに割り当てられた頂点は，それぞれ独立に拡張可能な頂点を求めるが，前方拡張を得るには $\mathcal{D}$ 内の近傍が $Q_{1,\cdots,p}$ に含まれていないことを，後方拡張を得るには Q_p 列からの近傍が $\mathcal{R}(P)$ 上に存在することを確認することで得られる．

4 プロトタイプの作成と実行効率の改善

文献 [5] では，3 章で述べたようなデータ構造を GPU 向けにどのように変換するかについての説明にとどめ，プログラム内部で具体的にどのように実装するかまでは記載されていなかった．本節では，実装レベルで述べられていないデータ構造やメモリアクセス方法について，より効率的な方法について検討し，得られた改善結果を述べる．

4.1 データ構造の改変

一般的なプログラミングでは多くのデータ構造に AoS(Array of Structures) が使用されることが多いが，CUDA プログラミングにおいては，4.2 節で述べる GPU メモリへのアクセスパターンの最適化などの観点から SoA(Structure of Arrays) を使用した方が高速になることが知られている．更に [13] により CUDA に適した，すなわち 8 バイト，もしくは 16 バイトでアライメントされた構造体であれば，SoA と比べても遜色のない速度が得られることも知られている．

DFS コードは 5 つ組のタプルであり，int 型で表すことが可能である．しかし，これを CUDA に適した 16 バイトでアライメントしようとしても $4byte(int) \times 5 = 20byte$ となり，パディングが大きくなってしまい少ない GPU メモリリソースを無駄にしてしまう．

本研究では，アルゴリズム 2 に示す関数で処理を開始する前に CPU 側で頻出 1-$edge$ パターンを求め，それ以降の拡張について GPU 側で処理を行うようにする．これにより，$k \geq 1$ の場合の a_k の $l_{i_k}(from\ label)$ は，$a_{k-i}(1 \leq i \leq k)$ のいずれかの $l_{j_{k-i}}(to\ label)$ に既に存在していることになる．従って，GPU メモリ上では $a_l = (i_l, j_l, l_{(i_l,j_l)}, l_{j_l})(1 \leq l \leq m)$ の 4 つ組のタプルで表すことが可能となり，$4byte(int) \times 4 = 16byte$ の無駄の無いアライメントが実現できる．

4.2 Read-Only キャッシュによる高速化

CUDA では隣接するスレッドが連続したメモリアドレスに並列にアクセスした際に，メモリアクセスがまとめられるコアレス (Coalesced) アクセスと呼ばれるアクセス方法となっており，GPU メモリへのアクセス時間が大幅に短縮されるため，データ構造をコアレスアクセス可能な形にすることが CUDA プログラム最適化において重要な項目となっている．しかし，グラフマイニングにおいては，例えば近傍の頂点ラベルを調べる際などにも常に連続したメモリアクセスを可能とするデータ構造へと変換することは大変に困難である．そこで本研究では，CUDA で利用可能な様々なキャッシュの内，Read-Only キャッシュを用いてメモリアクセスを行うようにした．Read-Only キャッシュとは，名の通り読み込み専用キャッシュであり，メモリアクセスが連続していなくても一定の高速化が期待できる．ただし，Read-Only キャッシュには，プリミティブ型，もしくは CUDA でシステム定義済みのアライメントされたベクトル型 (int2, int4, float2, float4 など) しか扱えないという制約があるため，データを置き換える必要がある．

データ構造をアライメントされたベクトル型に置き換え，Read-Only キャッシュを使用したところ，SoA よりも約 1.5-2 倍程の高速化に成功した．

5 評価実験

PGM では，様々な実データを用いた評価実験が行われた．本研究では，系統的なデータセットを用いて評価することを目的とする．なお，GPU はメモリ 4GB を搭載した GeForce GTX 970 を用いた．

5.1 実験データと実験結果

本研究では，グラフデータセット生成器である GraphGen[1] を用いて人工データを生成する．データセットの生成に際し，グラフ数，ユニークな頂点ラベル数，ユニークな辺ラベル数，ユニークな頂点・辺ラベル数に関するパラメータに着目し，表 2 に示すそれぞれ 5 つずつのデータを用意した．該当する項目以外のパラメータのデフォルト値は，表 1 に示すとおりである．また，グラフ数 50K のデータに限り，支持度の変化に関する実験も行った．

グラフ数 (†1)	10,000
頂点ラベルの種類数 (†2)	20
辺ラベルの種類数 (†3)	20
頂点・辺ラベルの種類数 (†4)	20
支持度 (%)(†5)	2

表 1: 各項目のデフォルト値

(†1)	10,000	30,000	50,000	70,000	90,000
(†2)	20	40	60	80	100
(†3)	20	40	60	80	100
(†4)	20	40	60	80	100
(†5)	0.2	0.35	0.5	0.65	0.8

表 2: 各項目ごとの 5 つのデータの設定値

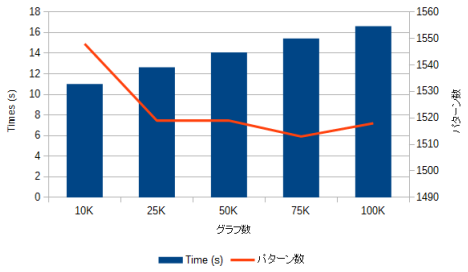

図 6: グラフ数別の実験結果

それぞれの実験結果（実行時間と得られたパターン数）を図 6-10 に示す．

5.2 考察

データベース内のグラフ数のみを変化させた結果である図 6 では，パターン数に関係なくグラフ数のみに

[1]http://www.cse.ust.hk/graphgen/

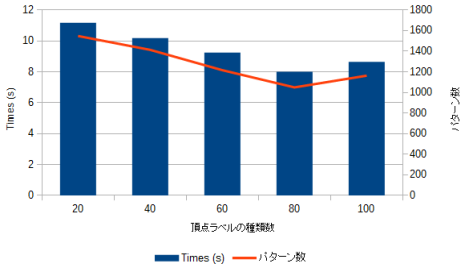

図 7: 頂点ラベルの種類数別の実験結果

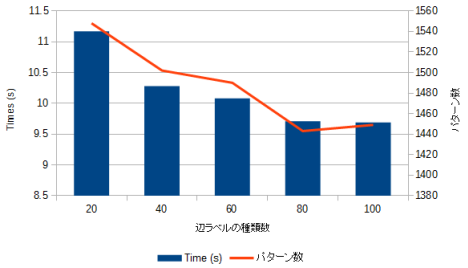

図 8: 辺ラベルの種類数別の実験結果

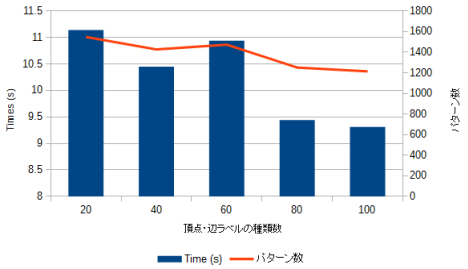

図 9: 頂点・辺ラベルの種類数別の実験結果

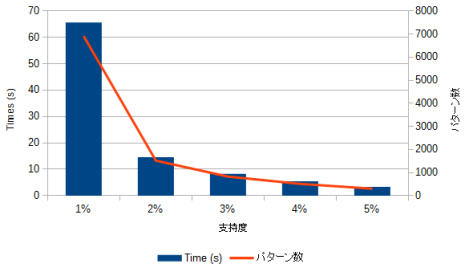

図 10: グラフ数 50K の支持度別の実験結果

比例して処理時間が長くなることが分かる．次に，頂点ラベルと辺ラベルの種類数による結果を考える．図 7, 8 より，処理時間とパターン数が比例していることが見て取れる．頂点・辺ラベルを同時に増やすと，図 9 より種類数が多くなるほどパターン数に対して処理時間が大幅に短縮している．グラフ数を 50K に固定した場合の 1〜5% の支持度別の結果（図 10）では，同様に処理時間とパターン数が比例しているが，2% から 1% に下げたとき，一気にパターンが増えてしまっている．

以上より，PGM アルゴリズムは頂点，辺ラベルの種類数が共に多いほど効率が良くなると言える．

6 まとめと今後の課題

本研究では，PGM を再実装し，データ構造を改変するとともに，適所にキャッシュを用いることで，ある程度の高速化を達成できた．また，系統的なテストデータを準備することで，PGM アルゴリズムの性質についても簡単な考察を行った．今後は，より多様な系統的なデータセットから，より多くの特徴を発見する必要があると考えられる．

参考文献

[1] X. Yan and J. Han : gSpan: Graph-Based Substructure Pattern Mining. *Proc. 2002 of Int. Conf. on Data Mining (ICDM'02)*, 2002.

[2] A. Inokuchi, T. Washio and H. Motoda : An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. *Proc. of the 4th European Conference on Principles of Data Mining and Knowledge Discovery, pp.13-23* 2000.

[3] M. Kuramochi and G. Karypis : Frequent Subgraph Discovery. *Proc. of the 2001 IEEE International Conference on Data Mining, pp.313-320*, 2001.

[4] S. Nijssen and J. N. Kok : A quickstart in frequent structure mining can make a difference. *Proc. of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining : pp.647-652*, 2004.

[5] R. Kessl, N. Talukder, P. Anchuri and M. Zaki : Parallel Graph Mining with GPUs. *Proc. of Journal of Machine Learning Research: Conference and Workshop, Vol.36, pp.1-36*, 2014.

[6] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki and A. Aboulnaga : Arabesque: A System for Distributed Graph Pattern Mining. *Proc. of the 25th ACM Symposium on Operating Systems Principles, pp.425-440*, 2015.

[7] M. A. Bhuiyan and M. A. Hasan : An iterative MapReduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering, Vol.27, Issue 3, pp.608-620*, 2015.

[8] W. Lin, X. Xiao and G. Ghinita : Large-scale frequent subgraph mining in MapReduce. *Proc. of 2014 IEEE 30th International Conference on Data Engineering (ICDE), pp.644-855*, 2014.

[9] F. N. Afrati, D. Fotakis, and J. D. Ullman : Enumerating subgraph instances using map-reduce. *Proc. of 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp.62-73*, 2013.

[10] S. Hill, B. Srichandan and R. Sunderraman : An Iterative MapReduce Approach to Frequent Subgraph Mining in Biological Datasets. *Proc. of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine pp.661-666*, 2012.

[11] X. Jiang, H. Xiong, C. Wang, and A. -H. Tan : Mining globally distributed frequent subgraphs in a single labeled graph. *Data & Knowledge Engineering, Vol.68, No.10, pp.1034-1058*, 2009.

[12] F. Wang, J. Dong and B. Yuan : Graph-Based Substructure Pattern Mining Using CUDA Dynamic Parallelism. *Proc. of the International Conference on Intelligent Data Engineering and Automated Learning, pp.342-349*, 2013.

[13] G. Mei and H. Tian : Performance Impact of Data Layout on the GPU-accelerated IDW Interpolation. *ArXiv e-prints*, 2014.