# Parallel Graph Mining with GPUs[*]

**Robest Kessl** KESSLR@CENTRUM.CZ
*Czech Technical University, Prague, Czech Republic*

**Nilothpal Talukder** TALUKN@CS.RPI.EDU
*Rensselaer Polytechnic Institute, Troy, NY*

**Pranay Anchuri** ANCHUPA@CS.RPI.EDU
*Rensselaer Polytechnic Institute, Troy, NY*

**Mohammed J. Zaki** ZAKI@CS.RPI.EDU
*Rensselaer Polytechnic Institute, Troy, NY*
*Qatar Computing Research Institute, Doha, Qatar*

**Editors:** Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

## Abstract

Frequent graph mining is an important though computationally hard problem because it requires enumerating possibly an exponential number of candidate subgraph patterns, and checking their presence in a database of graphs. In this paper, we propose a novel approach for parallel graph mining on GPUs, which have emerged as a relatively cheap but powerful architecture for general purpose computing. However, the thread-model for GPUs is different from that of CPUs, which makes the parallelization of graph mining algorithms on GPUs a challenging task. We investigate the major challenges for GPU-based graph mining. We perform extensive experiments on several real-world and synthetic datasets, achieving speedups up to 9 over the sequential algorithm.

**Keywords:** Parallel Frequent Graph Mining, Graphics Processing Unit

## 1. Introduction

Frequent graph mining has received a lot of attention in the recent past (Yan and Han, 2002; Inokuchi et al., 2000; Nijssen and Kok, 2004; Cook and Holder, 1994) with numerous applications, such as finding frequent substructures in biological networks or chemical compounds, similar communities in social networks, etc. Typically, graph mining requires one to generate all possible subgraph patterns and check subgraph isomorphisms from the patterns to the graphs in a database. Unfortunately, the number of subgraph patterns, even for a moderate number of vertices is exponentially large. Furthermore, subgraph isomorphism is known to be an NP-complete problem (Ullmann, 1976). Graph mining is computationally very hard, therefore, it is important to design scalable algorithms for this interesting problem.

In recent years, Graphics Processing Units (GPUs) have emerged as a cheap and relatively powerful computing architecture. A very large number of processor cores on GPU SIMT (single instruction multiple threads) architecture has made many compute-intensive tasks possible on commodity desktops. Parallel programming platforms, such as OpenCL

[*] This work was supported in part by NSF Awards CCF-1240646 and IIS-1302231.

and CUDA have enabled researchers to harness GPUs' computational prowess. However, GPUs are different from traditional CPUs, and are not suitable for all kinds of data structures and applications. Although GPUs have a large number of cores, unlike CPU-cores they are very compact and each core runs short-lived hardware threads in parallel. These threads are called kernels and they are launched from the CPU. Usually, applications with fine-grained parallelism and no communication among the parallel components are well-suited for GPUs. Given the limitations of GPU cores, the question naturally arises why we would choose GPUs for graph mining. The main reason is simply that most commodity computers already have relatively powerful GPUs and therefore provide an opportunity to harness them for performance gains, essentially for "free". GPUs are also generally cheaper than shared memory architectures with many cores or processors, and they require considerably less power than large shared memory machines.

Whereas several sequential algorithms for frequent graph mining have been proposed (Inokuchi et al., 2000; Kuramochi and Karypis, 2001; Yan and Han, 2002; Nijssen and Kok, 2004; Huan et al., 2003), parallel algorithms are not that prevalent. For instance, (Buehrer et al., 2006) presents parallel graph mining on multi-core CPUs. However, designing an effective graph mining algorithm on GPUs is challenging. One of the major bottlenecks is the amount of sequential administration required for the parallel steps. These include allocation of GPU memory and I/O transfers between the CPU and GPU, etc. Another challenge is that random memory accesses on GPU-memory may cause performance degradation. In this paper, we investigate and address all these challenges. In our GPU-based graph mining, we use the canonical labeling or ordering of graphs adopted in gSpan algorithm (Yan and Han, 2002), which efficiently prunes the duplicate subgraph patterns. Unlike in sequential approaches, in the GPU version, we compute all of the frequent extensions of a given pattern in parallel. Additionally, we show that having very fast parallel primitives like parallel scan (i.e., prefix sum), reduction, and sort, is very important for parallelization of a GPU-based algorithm. We describe how the data structures used by our algorithm are stored in the GPU memory for efficient access. Finally, we show the performance comparison of our algorithm and sequential graph mining using different real-world and synthetic datasets, showing significant speedups.

## 2. Background

In this section, we review some definitions and discuss related work. Let $V = 1, \cdots, m$ be the set of *vertices* and $E \subseteq V \times V$ the set of *edges*. A *graph* is defined as $G = (V, E, L)$, where $L$ is a labeling function. We denote by $L(v)$ the label for the vertex $v \in V$, and by $L(u, v)$ the label for the edge $(u, v) \in E$.

**Subgraph Isomorphism**: A graph $G_1 = (V_1, E_1, L_1)$ is *subgraph isomorphic* to $G_2 = (V_2, E_2, L_2)$, denoted as $G_1 \simeq G_2$, if there exists a injective function, $f : V_1 \rightarrow V_2$ such that: 1) $\forall u \in V_1$, $L_1(u) = L_2(f(u))$, and 2) $\forall (u, v) \in E_1$, $(f(u), f(v)) \in E_2$ such that $L_1(u, v) = L_2(f(u), f(v))$.

**Support**: Let $\mathcal{D} = \{G_1, \cdots, G_n\}$ be a database of graphs. The support of a pattern $P$ is defined as the number of graphs in $\mathcal{D}$ that contain a subgraph isomorphic to $P$, i.e., $|\{G_i : G_i \in \mathcal{D} \text{ and } P \simeq G_i\}|$. Further, an occurrence of a subgraph pattern in the graph database corresponds to an isomorphism, and is called an *embedding*. Fig. 1 shows an example database of 3 graphs (with edge labels omitted). The support of $P$ is $\sigma(P) = 2$ since

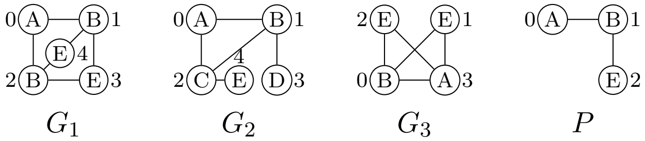


Figure 1: Graph database, $\mathcal{D}$ and pattern, $P$

$P$ is a subgraph of $G_1$ and $G_3$. A pattern graph is called frequent if $\sigma(P) \geq minsup$, where $minsup \in \mathbf{Z}$ is a user-specified *minimum support threshold*. For example, if $minsup = 2$, then $P$ in Fig. 1 is frequent. Given $\mathcal{D}$ and $minsup$ the frequent subgraph mining task is to find all $P$ such that $\sigma(P) \geq minsup$.

**DFS Code**: For a pattern $P = (V_P, E_P)$, and given a DFS (depth-first) tree over the nodes of $P$, the DFS code for $P$ with respect to the given tree is a list of five-tuples in DFS order of the edges. Each five-tuple describes an edge $(v_i, v_j, l_i, l_{ij}, l_j)$ such that $v_i, v_j \in V_P, (v_i, v_j) \in E_P$, $l_i = L(v_i)$, $l_j = L(v_j)$ and $l_{ij} = L(v_i, v_j)$. A DFS code of $P$ is therefore given as $S = (s_1, \dots, s_{|E_P|})$, where $s_\ell = (v_i, v_j, l_i, l_{ij}, l_j)$. For example, in Fig. 1 we can represent $P$ as the DFS code $(0, 1, A, -, B)(1, 2, B, -, E)$. Here, '$-$' denotes an empty edge label. Another example of DFS code for $P$ is $(0, 1, B, -, A)(0, 2, B, -, E)$. There can be exponentially many DFS codes for the same graph due to different DFS traversals and automorphisms of $P$. However, one can define a precedence relation $\prec$ on them, so that we obtain a total order on the DFS codes; the unique/minimum element is called the *canonical* or *minimal* DFS code for $P$, and is denoted as $minDFS(P)$. For example, $minDFS(P) = (0, 1, A, -, B)(1, 2, B, -, E)$. For the detailed ordering relation $\prec$ of DFS codes, see Yan and Han (2002). Instead of $minDFS$ one can also use minimal adjacency matrices to define a unique representative for the automorphism group of $P$ (Kuramochi and Karypis, 2001; Inokuchi et al., 2000; Huan et al., 2003).

**Rightmost Path and Edge Extension**: The search for frequent patterns is usually done in a breadth-first or depth-first manner, starting with single edge graphs and adding an extra edge at each level or step, respectively. The use of $minDFS$ allows one to prune duplicate/isomorphic patterns. However, to generate new candidates for support computation, we have to consider two types of edge extensions in a DFS code: 1) a *forward edge* (when $v_i < v_j$) introduces a new vertex in the DFS code, and 2) a *backward edge* (when $v_i > v_j$) is between two existing vertices in the DFS code, resulting in a cycle. Define the rightmost path $\mathcal{R}(P)$ of a pattern $P = (V_P, E_P)$ as the shortest path from $v_1 \in V$ to the rightmost vertex $v_{|V_P|} \in V_P$ in $minDFS(P)$, i.e., $v_{|V_P|}$ is the rightmost child in the corresponding DFS tree for $P$. Forward edge extensions are allowed only from the vertices on the rightmost path $\mathcal{R}(P)$, and the backward extensions are allowed only from the vertex, $v_{|V_P|}$ to the rest of the vertices on $\mathcal{R}(P)$; it is guaranteed that all possible candidate patterns will be generated (Yan and Han, 2002).

**Related Work:** Frequent graph mining is a well studied problem (Cook and Holder, 1994; Huan et al., 2003; Inokuchi et al., 2000; Kuramochi and Karypis, 2001; Nijssen and Kok, 2004; Yan and Han, 2002; Chaoji et al., 2008). One of the earliest algorithms, SUBDUE (Cook and Holder, 1994), searches for the potential frequent substructures and their ability to compress the graph database. The later approaches to the problem use systematic generation of candidate subgraph patterns. Methods such as AGM (Inokuchi et al., 2000) and FSG (Kuramochi and Karypis, 2001) use the level-wise growth of candidate patterns.

However, these approach can generate a lot of duplicate candidate patterns, making them inefficient. Instead, algorithms such as gSpan (Yan and Han, 2002), Gaston (Nijssen and Kok, 2004) and FSM (Huan et al., 2003) use canonical ordering of the patterns to avoid duplicates. Further, Gaston generates simple patterns like paths and trees first before handling full graphs (i.e., cycles). DMTL (Chaoji et al., 2008) is a generic framework for frequent pattern mining that can handle different pattern types from itemsets to graphs. A parallel implementation of gSpan on shared memory architecture was described in (Buehrer et al., 2006), evaluating both static and dynamic task allocation. However, this parallelization strategy cannot be applied to the GPU SIMT architecture as the threads are short-lived and have many other limitations. A recent work (Lin et al., 2014) tackles the graph mining problem using the mapreduce framework. We are aware of only one other work for graph pattern mining on GPUs (Wang et al., 2013), which uses the dynamic parallelism supported by the newer NVIDIA K20 GPU. They report speedups of 10-15, however, they used only one very small dataset (422 chemical compounds with 27 avg. nodes and 28 avg. edges per graph). Our results in Section 8 suggest that their reported speedups may be too optimistic.

## 3. Graph Mining on GPUs

Our GPU-based graph mining combines the best features of state-of-the-art graph mining algorithms like gSpan (Yan and Han, 2002), Gaston (Nijssen and Kok, 2004) and DMTL (Chaoji et al., 2008). First, it employs the canonical ordering of graphs, called the DFS-code, from gSpan, which efficiently prunes duplicate subgraph patterns. Second, like Gaston/DMTL, it stores all of the isomorphisms/embeddings for each pattern, which leads to fast support computation (subgraph isomorphism testing), and is well-suited for parallelizing on GPUs. Note that this is in contrast to the original gSpan algorithm that performs subgraph isomorphism from scratch for each new pattern. A major advantage of using embedding lists is that each embedding of a subgraph pattern can be processed independently in parallel, which is conducive for GPU-based parallelization. On the other hand, unlike these sequential methods, the GPU algorithm computes all the frequent extensions (and their embeddings) of a given pattern *in parallel*, instead of one at a time.

The pseudo-code for GPU-based graph mining is shown in Algorithm 3, with the computationally expensive steps executed on the GPU as shown. Let $\Sigma_{\mathcal{D}}(P)$ denote the embedding list for the pattern $P$ in database $\mathcal{D}$, i.e., the set of all the isomorphisms from $P$ to graphs in $\mathcal{D}$. Initially the method is called with the pattern $P = \emptyset$, and all the canonical single edge graphs are found. The algorithm always extends the canonical DFS representation of an input $P$ with an additional edge. It searches for the edges that are incident with the rightmost path in each of the embeddings of $P$. We denote these edge extensions by $\mathcal{E}_{\mathcal{D}}(P)$, which comprise the candidate extensions from $P$. Thus, given an input frequent pattern $P$, the extend method first finds its extensions (Line 1), and then prunes out the infrequent ones after computing their support (Line 2). Only those candidates that have the minimal DFS are recursively extended (Line 5).

Those steps that are performed in parallel on the GPU are as marked. In brief, the parallel steps include the search for all extensions of the current pattern $P$ (Line 1), support computation for all the extensions and removal of the infrequent ones (Line 2), and the creation of embedding lists for each extension (Line 7). These steps involve efficient processing in parallel of a very large number of extensions or embeddings in each iteration.

**Algorithm 1** Graph Mining on GPUs (Database $\mathcal{D}$, Threshold *minsup*, Pattern $P$, Embeddings $\Sigma_{\mathcal{D}}(P)$)

```
//Initial Call: $P = \emptyset$, $\Sigma_{\mathcal{D}}(P) = \emptyset$
 1: (GPU step) Get all possible edge extensions $\mathcal{E}_{\mathcal{D}}(P)$ of $P$
 2: (GPU step) Compute support for all extensions in $\mathcal{E}_{\mathcal{D}}(P)$ and remove infrequent extensions
 3: for each $e = (v_i, v_j, l_i, l_{ij}, l_i) \in \mathcal{E}_{\mathcal{D}}(P)$ do
 4:     $P' \leftarrow P$ extended by $e$
 5:     if $P' = minDFS(P')$ then
 6:         output $P'$
 7:         (GPU step) Create $\Sigma_{\mathcal{D}}(P')$
 8:         GraphMining($\mathcal{D}, minsup, P', \Sigma_{\mathcal{D}}(P')$)
 9:     end if
10: end for
```

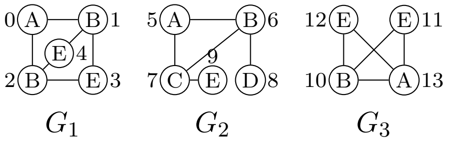
It is important to note that unlike gSpan, we also perform support computation before minimal DFS code checking, since we compute the support for many extensions in parallel. However, DFS code minimality checking (Line 5) is not done on the GPU since there isn't enough parallelism to exploit when checking whether a single pattern is minimal or not. On the other hand, checking minimality for all extensions simultaneously (in parallel) would require too much memory.

Due to a different thread model, programming GPUs is quite different from traditional CPU-based multi-threaded or multi-core approaches. In brief, the major challenges are as follows: *(1) Ensuring efficient access of data*: In sequential graph mining, the graph edges and the pattern embeddings are usually stored in main memory using efficient data structures (e.g., hashmap or linked lists). However, we cannot take advantage of these on GPUs. We need to store the data in GPU global memory where random memory access may cause significant slowdown. GPU global memory is not cached and the consecutive memory locations are read in blocks by multiple GPU threads. Therefore, we need to serialize the data, fit them in GPU memory, and ensure access locality, or "coalesce" global memory access. *(2) Dynamic memory allocation and I/O cost*: GPU threads or kernels are launched from the CPU and they access data from the GPU memory. We need to perform dynamic memory allocation on GPUs and transfer the data from the main memory. After the kernel exits the results are transferred back to the CPU for further processing. Thus, the major bottlenecks for GPU applications are the dynamic memory allocation on GPUs, and the data transfers between the main-memory and the GPU-memory. We need to avoid the costly dynamic allocations and I/O transfers as much as possible. *(3) Parallelization with GPU threads*: Each GPU thread can perform a fine-grained task, and thus algorithms perform better when threads do not need to communicate. For example, the support computation (Line 2) for each new extension can be done in parallel. GPUs have many processor cores, and one of their advantages is that parallel primitives, such as scan, reduction, etc., can be efficiently implemented. The key for a high performance GPU-based algorithm is to transform a parallelizable step into a set of fast parallel primitives. Below we describe in detail how we address these challenges.

## 4. GPU Data Structures

We now describe how the graph database and pattern embeddings are stored in GPU global-memory.

**Storage of the Graph Database:** To ensure efficient access by the GPU threads we serialize the graph database $\mathcal{D}$ in GPU global memory such that we can quickly lookup the neighborhood of any given vertex, which is required for computing the set of edge extensions from each of the embeddings of $P$. We also need to quickly determine the graph id of a vertex during support computation. Each vertex in the serialized database has a unique number that identifies the vertex in the whole database, called *global vertex id*, and we consider the whole database $\mathcal{D}$ as one large (disconnected) graph $G^{\mathcal{D}} = (V^{\mathcal{D}}, E^{\mathcal{D}})$ comprising the individual graphs $G_i = (V_i, E_i)$. The maximal number of vertices over the database graphs is stored in $m_V = \max_i |V_i|$, and the vertices of the $i$-th graph are re-numbered so they lie in the interval $\big[i \cdot m_V, (i+1) \cdot m_V\big)$. In Fig. 2 the graph database of Fig. 1 is renumbered with the *global vertex id*s.



$(a)$ Graph database with *global vertex ids*

| *global id* | 0 | | 1 | | | 2 | | | 3 | | 4 | | 5 | | 6 | | | 7 | | | 8 | 9 | 10 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **N** | 1 | 2 | 0 | 3 | 4 | 0 | 3 | 4 | 1 | 2 | 1 | 2 | 6 | 7 | 5 | 7 | 8 | 5 | 6 | 9 | 6 | 7 | 11 | 12 | 13 |

| *global id* | 11 | | 12 | | | 13 | | |
|---|---|---|---|---|---|---|---|---|
| **N**(cont'd) | 10 | 13 | 10 | 11 | 12 | 10 | 11 | 12 |

| *index* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **O** | 0 | 2 | 5 | 8 | 10 | 12 | 14 | 17 | 20 | 21 | 22 | 25 | 27 | 30 | -1 |

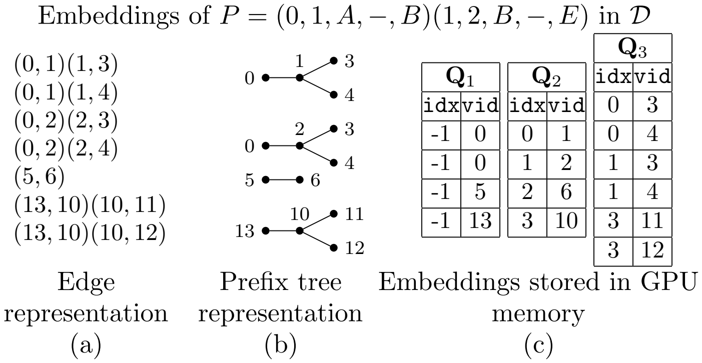
$(b)$ GPU representation (partial) of graph database

Figure 2: Graph database with *global vertex id*s and its storage in GPU memory.

The neighborhoods of all vertices are stored in an array $\mathbf{N}$, where each $\mathbf{N}[l]$ indicates a *global vertex id*. The offsets into $\mathbf{N}$ are stored in an array $\mathbf{O}$ of size $|\mathcal{D}| \cdot m_V$, i.e., the offset of the neighborhood of $i$-th vertex in $\mathbf{N}$ is stored in $\mathbf{O}[i]$. The elements representing invalid vertices in $\mathbf{O}$ are filled with -1. Finally, the labels of vertices and edges are stored in two separate arrays of sizes $|\mathcal{D}| \cdot m_V$ and $|\mathbf{N}|$, respectively. In Fig. 2, the arrays $\mathbf{N}$ and $\mathbf{O}$ for the example graph database are shown, e.g., node 0 has two neighbors, 1 and 2, which comprise the first two elements of $\mathbf{N}$, and node 1 has neighbors 0, 3, and 4, which comprise the next three elements. Note that the global id row is shown only for convenience, since the offsets into $\mathbf{N}$ are specified by $\mathbf{O}$. As we can see, the neighborhood information is stored in consecutive memory locations which ensures efficient lookup for any GPU thread. Additionally, we can determine the graph id from the vertex id (by simply computing the modulo function using $m_V$).

**Storage of Embeddings:** A pattern $P$ can be located at many positions in a graph, $G \in \mathcal{D}$. Let $G' \simeq G$ be a subgraph of $G$, with $G'$ being isomorphic to $P$. In other words $G'$ is an embedding of $P$ in graph $G$. Let the minimal DFS code of $P$ be $minDFS(P) = (c_1, \ldots, c_{|E_P|})$ and the set of edges in $G'$ be $(d_1, \ldots, d_{|E_P|})$, where each $c_k$ corresponds to $d_k$, so that the isomorphism is preserved. Both $c_k$ and $d_k$ can be described as a five-tuple $(v_i, v_j, l_i, l_{ij}, l_j)$. The vertices, $v_i$ and $v_j$ of the embedding $G'$ are *global vertex id*s, whereas for the pattern $P$ they indicate the *pattern vertex numbers* used in $minDFS(P)$.

Fig. 3 shows the embeddings for the pattern $P = (0, 1, A, -, B)(1, 2, B, -, E)$ (from Fig. 1). Fig. 3(a) shows the edge representation. The edges here are shown as $(v_i, v_j)$ where $v_i$ and $v_j$ correspond to the *global vertex id*s. For example, the first isomorphism for the pattern is given as $f(0) \to 0$, $f(1) \to 1$ and $f(2) \to 3$, which corresponds to the edges $(0,1),(1,3)$ in the graph database (using global ids). The number of pattern embeddings in $\mathcal{D}$ can be very large due to the isomorphisms and automorphisms of the pattern. Since different embeddings can share the same set of edges, a compact way to list them is in the form of *prefix trees* as shown in Fig. 3(b). Furthermore, in order to store them in the limited GPU-memory, we use a compact representation of the pattern embeddings by linearizing the prefix trees as a sequence of vertices.



Embeddings of $P = (0,1,A,-,B)(1,2,B,-,E)$ in $\mathcal{D}$

$(0,1)(1,3)$
$(0,1)(1,4)$
$(0,2)(2,3)$
$(0,2)(2,4)$
$(5,6)$
$(13,10)(10,11)$
$(13,10)(10,12)$

Edge representation
(a)

Prefix tree representation
(b)

| $\mathbf{Q}_1$ | |
|---|---|
| idx | vid |
| -1 | 0 |
| -1 | 0 |
| -1 | 5 |
| -1 | 13 |

| $\mathbf{Q}_2$ | |
|---|---|
| idx | vid |
| 0 | 1 |
| 1 | 2 |
| 2 | 6 |
| 3 | 10 |

| $\mathbf{Q}_3$ | |
|---|---|
| idx | vid |
| 0 | 3 |
| 0 | 4 |
| 1 | 3 |
| 1 | 4 |
| 3 | 11 |
| 3 | 12 |

Embeddings stored in GPU memory
(c)

Figure 3: Embeddings Representations for $P$

The compact GPU representation for our running example is shown in Fig. 3(c). Given a pattern $P$ with $p$ vertices, its serialized embeddings comprise $p$ columns, $|\mathbf{Q}_1|, \cdots, |\mathbf{Q}_p|$. Each element in the column consists of a pair $(\texttt{idx}, \texttt{vid})$, where `idx` is an index pointing to the vertex of the same embedding in the previous column, and the `vid` is the *global vertex id*. The entries in the last column, $\mathbf{Q}_p$, actually represent headers of different linked lists. We can reconstruct an embedding by following `idx` from $\mathbf{Q}_p$ for one particular element into the previous column $\mathbf{Q}_{p-1}$, and continuing in the same manner until we reach $\mathbf{Q}_1$. The `idx` of $\mathbf{Q}_1[i]$ contains -1, marking the end of the chain. Thus, by following the back-links from the last column, we can reconstruct the whole set of embeddings. For example, in Fig. 3, we can trace back from the entry at index 2 of $\mathbf{Q}_3$, namely $vid = 3$ with $idx = 1$ to go to the entry at index 1 in $\mathbf{Q}_2$, which has $vid = 2$ and $idx = 1$. Next, we look up index 1 in $\mathbf{Q}_1$, which has $vid = 0$ and $idx = -1$, which marks the end of the trace. The isomorphism is the sequence of *vid*s we obtained above, i.e., $f(2) \to 3$, $f(1) \to 2$, and $f(0) \to 0$, which corresponds to the embedding $(0,2)(2,3)$ in the edge representation. The example also shows remnants of invalid embeddings during the mining process. For instance, the embeddings of the first pattern edge $(0,1,A,-,B)$ in the graph database include $(0,1)$, $(0,2)$, $(5,6)$, $(13,10)$. Among these the mapping to $(5,6)$ cannot be extended by the second pattern edge $(1,2,B,-,E)$. Whereas this remnant exists in $\mathbf{Q}_1$, and $\mathbf{Q}_2$, it will be automatically discarded since we cannot trace it back from any embedding in $\mathbf{Q}_3$.

During pattern extension, for storing an additional *forward edge*, we need to add another column $\mathbf{Q}_{p+1}$. The reason is that one vertex of the new edge must always be present in the embeddings. On the other hand, we do not need to store the *backward edges* in the embedding columns since those can be obtained from the DFS code of $P$. We also do not store the graph id as it can be computed from *global vertex id*.

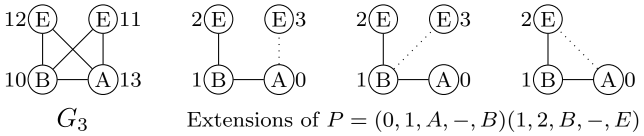
## 5. Pattern Enumeration: Embedding Extension

We now discuss how we parallelize the pattern extension step in Algorithm 3 (Line 1). The set of all one-edge extensions of the canonical DFS code for $P$ comprise the candidate patterns for the next level. Since we store the pattern embeddings, we can lookup the neighborhood of the embedded vertices in the database to get only the relevant extensions, i.e., $\mathcal{E}_{\mathcal{D}}(P)$. The valid extensions are possible only from the vertices on the rightmost path, $\mathcal{R}(P)$, and they can be obtained independently of one another. This makes it very suitable for parallelization using GPU threads as there is no communication required.

**Algorithm 2** Get-Extensions (Database $\mathcal{D}$, Embedding columns $\mathbf{Q}_k$ on $\mathcal{R}(P)$, $1 \leq k \leq p$)

```
1: (GPU step) Compute max degree $m$ of vertices in $\mathbf{Q}_k$.
2: Allocate array $\mathbf{V}$ of size $m \times |\mathbf{Q}_k|$ to indicate valid extensions from a vertex; initialize with 0s.
3: (GPU step) Look-up the neighborhood of each vertex in $\mathbf{Q}_k$, and set the corresponding entry in
   $\mathbf{V}$ to 1 for a valid forward/backward extension.
4: (GPU step) Perform exclusive scan of $\mathbf{V}$ to produce indexes for a new valid extensions array.
5: Allocate array $\mathbf{EXT}_k$ to store valid extensions from $\mathbf{Q}_k$.
6: (GPU step) Store the extensions in $\mathbf{EXT}_k$ using index array computed in step 4.
7: return $\mathbf{EXT}_k$
```

Let us assume that the pattern $P$ has $p$ vertices. As described in Section 4, we store the embeddings of these $p$ vertices as $p$ columns, $\mathbf{Q}_k, 1 \leq k \leq p$. Therefore, we have $|\mathbf{Q}_p|$ embeddings that need to be processed. $\mathcal{R}(P)$ comprises the set of indexes for the columns $\mathbf{Q}_k$ that contain vertices on the rightmost path. Algorithm 5 specifies the steps to obtain the extensions of $P$. Each vertex on the rightmost path of an embedding is assigned to a thread. The algorithm finds all the valid extensions of an embedding column $\mathbf{Q}_k$ in parallel, assuming $\mathbf{Q}_k$ is on $\mathcal{R}(P)$, with the GPU steps as indicated. The basic idea of the algorithm is that each GPU thread performs a lookup of the neighborhood of its assigned vertex (Step 3) and later only the valid forward/backward extensions are extracted into a new extensions array called $\mathbf{EXT}_k$ (Steps 4 to 6), which is used for support computation. As an example, the valid forward and backward extensions from the pattern $P = (0, 1, A, -, B)(1, 2, B, -, E)$ in the example graph $G_3$ are shown in Fig. 4($a$). The last one is a backward extension (since it introduces a cycle).



($a$) Examples of forward/backward extensions

| | | |
|---|---|---|
| $1_{(0)}$ | $7_{(1)}$ | $13_{(3)}$ |
| $2_{(0)}$ | $8_{(1)}$ | $14_{(4)}$ |
| $3_{(0)}$ | $9_{(2)}$ | $15_{(3)}$ |
| $4_{(0)}$ | $10_{(2)}$ | $16_{(4)}$ |
| $5_{(13)}$ | $11_{(10)}$ | $17_{(11)}$ |
| $6_{(13)}$ | $12_{(10)}$ | $18_{(12)}$ |

($b$) Threads are allocated column-wise. Table entries: *thread-id*$_{(vertex\text{-}id)}$

Figure 4: Example of valid extension and thread allocation

**Assignment of the Threads**: The linearized prefix tree representation of the embeddings can be virtually treated as a matrix where each row indicates the vertices in an embedding, and each column corresponds to a vertex in $P$. To obtain the extensions, each vertex on

$\mathcal{R}(P)$ is assigned to one thread, having a total of $|\mathcal{R}(P)| \times |\mathbf{Q}_p|$ threads for all embedding columns. For a valid *forward extension* each thread has to check that the new *to* vertex is not already present in the embedding. The valid *backward extensions* are only obtained for the vertices in the last column, and the thread has to check that the *to* vertex is present on the rightmost path, $\mathcal{R}(P)$.

To ensure the access locality on GPU, the threads are allocated in *column-major* order of the embeddings. Fig. 4($b$) shows the thread id and the vertex id assignment of our running example in matrix form. The vertex ids are shown as subscripts in brackets. For instance there are six possible embeddings of $P$; we can see that the thread ids are assigned in column-major order. Each thread is tasked to find extensions from its assigned vertex on the rightmost path. The thread actually allocates its rightmost path vertex by computing the assigned column and following the back-links. The threads with numbers $1, \ldots, |\mathbf{Q}_p|$ are assigned to the first vertex on the rightmost path, threads $|\mathbf{Q}_p| + 1, \ldots, 2 \cdot |\mathbf{Q}_p|$ to second vertex on the rightmost path, and so on. Since we assign the threads in column-major fashion there is a sequence of threads that process the same vertex. For instance, in Fig. 4($b$) the first four threads in the first column process vertex 0. Although the vertex logically belongs to different embeddings, during the processing of the vertex the threads read the same neighborhood, i.e., the same memory locations. This gives us a good GPU memory access pattern, since the GPU runs blocks of 32 threads at once and they can efficiently access memory locations in one fetched block.

**Storage of the Extensions**: The extensions found by the threads are stored in an array, **EXT**. The extensions are organized in segments, so that the extensions from $k$-th vertex on the rightmost path $\mathcal{R}(P)$ are stored in segment $\mathbf{EXT}_k$. Each element of **EXT** contains: 1) the DFS code, $(v_i, v_j, l_i, l_{ij}, l_j)$, where $v_i, v_j$ are pattern vertex ids and 2) the embedding description of the edge extension, which consists of: a) the *global vertex id* of the *from* vertex, $v_i^g$, b) the *global vertex id* of the *to* vertex, $v_j^g$, and c) the `row` pointer to the *from* vertex in the last embedding column $\mathbf{Q}_p$.

The storage of the extensions is performed in the following manner as shown in Algorithm 5: 1) We compute the maximum degree $m$ for every vertex on $\mathcal{R}(P)$ (Step 1). Then we use an array $\mathbf{V}$ of size $m \times |\mathbf{Q}_p| \times |\mathcal{R}(P)|$ to indicate the valid extensions of the vertices on $\mathcal{R}(P)$ (Step 2). Each entry in $\mathbf{V}$ can represent three different values: a forward extension, a backward extension and no extension. For simplicity we omit the details. 2) We then perform the parallel exclusive scan of $\mathbf{V}$ (Step 4), which gives the indexes of the valid extensions from $\mathbf{V}$ into **EXT**. The exclusive scan stores at the $i$-th index the sum of elements from index 0 to $i-1$ of the input array, i.e., prefix sum excluding the $i$-th element. 3) Finally in Step 6, we store the DFS code and the embedding description of an extension in **EXT** at the positions looked-up in the scan results.

Extensions:

| $k$ | $\mathbf{EXT}_0$ | | | | | | | $\mathbf{EXT}_1$ | | | | $\mathbf{EXT}_2$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $v_i^g$ | 0 | 0 | 0 | 0 | 5 | 13 | 13 | 1 | 1 | 6 | 10 | 3 | 4 | 11 | 12 |
| $v_j^g$ | 2 | 2 | 1 | 1 | 7 | 11 | 12 | 3 | 4 | 8 | 12 | 2 | 2 | 13 | 13 |
| $l_j$ | B | B | B | B | C | E | E | E | E | D | E | B | B | A | A |
| **B** | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| scan **B** | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 1 |

Support Computation:

| | $(0,B)$ | | | $(0,C)$ | | | $(0,E)$ | | | $(1,E)$ | | | $(1,D)$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **F** | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| scan of **F** | 1 | | | 1 | | | 1 | | | 2 | | | 1 | | |

Figure 5: Extensions of $P = (0, 1, A, -, B)$ $(1, 2, B, -, E)$, and support computation of the extensions from vertices 0 and 1. Array **B** contains 1 at graph boundaries within each segment.

Fig. 5 shows the segments consisting of the valid extensions from the vertices 0, 1 and 2 from $P$. The *global vertex id* of the *to* vertex, i.e., $v_j^g$ and its label $l_j$ are shown for each valid extension. However, we do not show the `row` pointers to avoid clutter. For example, consider the renumbered graph database from Fig. 2, and consider the three distinct extensions of $P$ in graph $G_3$ shown in Fig. 4(*a*). Consider the embedding $(13,10)(10,11)$ for the pattern edges $(0,1)(1,2)$. Here vertex 0 is mapped to $f(0) \rightarrow 13$, and there is only one valid forward extension possible, namely from 13 to 12. Likewise, from the second embedding $(13,10)(10,12)$ there is one forward extension, from 13 to 11. These two graph vertices, namely 11 and 12, appear in the last two columns under the extensions for $k = 0$, $\mathbf{EXT}_0$. Note that a backward extension is possible only from the rightmost vertex 2 in $P$. For example, for $G_3$, both of the embeddings above have 13 as the *to* vertex, as shown under $\mathbf{EXT}_2$. The other entries for the all the database graphs can be obtained in a similar manner.

**Properties of EXT**: It is important to note that the threads with the ids $i$ and $i+1$ store the extensions in adjacent memory locations in **EXT**. This order of storage gives us some important properties of **EXT** which are as follows: 1) The extensions from a vertex in $\mathcal{R}(P)$ are stored in consecutive locations and they form a segment of extensions. In other words, the extensions with the same $v_i$ in the DFS code constitutes a segment. We denote $k$-th segment by $\mathbf{EXT}_k$. 2) Additionally, in each segment $\mathbf{EXT}_k$ there are blocks of elements with the *global vertex ids*, $v_i^g$'s and $v_j^g$'s corresponding to the same database graph $G \in \mathcal{D}$. We will see in the next section that these properties are very useful for efficient support computation.

## 6. Support Computation

Having obtained all the edge extensions of a pattern $P$ from the database, we next compute their support in parallel and discard the infrequent extensions. The segment $\mathbf{EXT}_k$ contains all extensions from a vertex on the rightmost path, and thus the segments can be processed independently of each other. We do this in two GPU steps: 1) extraction of the unique extensions from **EXT**, and 2) support computation of the extensions, as detailed next.

**Extract Extensions:** We extract *forward extensions* from all segments, $\mathbf{EXT}_k, 1 \leq k \leq |\mathcal{R}(P)|$, and *backward extensions* only from the last segment, i.e., $\mathbf{EXT}_{|\mathcal{R}(P)|}$. Recall that an element in **EXT** consists of the DFS code edge $(v_i, v_j, l_i, l_{ij}, l_j)$ and the embedding description. For a *forward extension* in $\mathbf{EXT}_k$ the DFS extension $(v_i,v_j)$ and the label of the *from* vertex, $l_i$, remain the same. Let $\mathcal{L}_V$ be the set of all vertex labels and $\mathcal{L}_E$, the set of all edge labels. For the sake of simplicity, we also assume that $l_i$ and $l_j$ are chosen from the integer range $[1, |\mathcal{L}_V|]$. Similarly, the range of the edge label $l_{ij}$ is $[1, |\mathcal{L}_E|]$. We observe that there are at most $|\mathcal{L}_E| \times |\mathcal{L}_V|$ *forward extensions* in $\mathbf{EXT}_k$ for different $l_{ij}$ and $l_j$'s. Also, there can be at most $|\mathcal{L}_E| \times (|\mathcal{R}(P)| - 1)$ *backward extensions*. This is because in this case the $v_j$'s are chosen only from $\mathcal{R}(P)$, so there can be at most $|\mathcal{R}(P)| - 1$ different $l_j$'s (excluding $l_i$).

To obtain the *forward extensions* we allocate an array of size $|\mathcal{L}_V| \times |\mathcal{L}_E|$ initialized with 0s. Each GPU thread looks at the DFS code of an $\mathbf{EXT}_k$ element and marks the array at index $(l_{ij} \times |\mathcal{L}_V| + l_j)$ with 1. Multiple threads may write to the same location, and at least one of them will succeed. Finally, we extract the DFS codes of the marked locations into another array using parallel primitives. The *backward extensions* can also be obtained in a similar fashion. For example, for the embedding 13 ← 10 ← 12 shown in Fig. 3, there is a *backward edge* in Fig. 5 going from the *global vertex id* 12 to 13 (in the extensions from segment 2). The DFS code of this extension is $(2, 0, E, -, A)$. The rest of the examples in Fig. 5 are forward extensions.

**Algorithm 3** Compute-Support (Extension array $\mathbf{EXT}_k$, Extension $e = (v_i^g, v_j^g, v_i, v_j, l_i, l_{i,j}, l_j)$

```
Allocate array $\mathbf{B}$ that stores the boundaries of block of extensions of one graph in $\mathbf{EXT}_k$.
(GPU step) Fill $\mathbf{B}$ using graphid($\mathbf{EXT}_k[i]$).
(GPU step) Exclusive scan of $\mathbf{B}$ that computes the index of each block in $\mathbf{EXT}_k$.
Allocate array $\mathbf{F}$ and fill with 0.
(GPU step) Store 1 in $\mathbf{F}$ if $e$ matches an element in $\mathbf{EXT}_k$ using index from $\mathbf{B}$.
(GPU step) Compute support $s$ by reduction of $\mathbf{F}$.
return $s$
```

**Support Computation of the Extensions:** Consider the support computation for a single DFS code extension that belongs to the segment, $\mathbf{EXT}_k$. From the properties of the $\mathbf{EXT}$ described in Section 5 it follows that the segment $\mathbf{EXT}_k$ consists of blocks of extensions, each corresponding to some graph in $\mathcal{D}$. However, the number of such blocks or graphs in $\mathbf{EXT}_k$ can be much smaller than $|\mathcal{D}|$. Since the DFS code extension can only be present in those graphs, we remap the database graph ids for $\mathbf{EXT}_k$. Let the number of graphs in $\mathbf{EXT}_k$ be $g$, with the graph ids remapped into the range $[1, g]$. Briefly, the remapping is done by the following steps: 1) allocating an array $\mathbf{B}$ of size $|\mathbf{EXT}_k|$ initialized with 0s; 2) storing 1 at the graph boundaries, i.e., at the end of each block; and 3) finally, performing an exclusive scan on $\mathbf{B}$. Now, $\mathbf{B}$ contains a mapping of the sparse database graph ids into a continuous range (see Algorithm 6, Steps 1–3). Fig. 5 shows the $\mathbf{B}_{oundary}$ array and the result of the exclusive scan, which gives the remapped graph ids. For example, $\mathbf{EXT}_2$ contains only two graphs, with remapped graph id 0 (vertices 1 and 2) and 1 (vertex 13). Note that the remapping can be re-used for computation of support of all other extensions, amortizing the time spent in this step.

The support computation of the extension (see Algorithm 6, Steps 4–6), proceeds as follows. In Algorithm 6 we use graphid($\mathbf{EXT}_k[i]$) to denote the graph id of the $i$-th extension of the array $\mathbf{EXT}_k[i]$. First, we allocate a flag array $\mathbf{F}$ of size $g$ and initialize it with 0s. Then, we execute $|\mathbf{EXT}_k|$ threads, where each thread $t_i$ looks into $\mathbf{EXT}_k[i]$ and checks if its DFS code corresponds to the DFS extension in question. If the DFS code matches, the thread performs a lookup on $\mathbf{B}[i]$ and stores 1 in $\mathbf{F}[\mathbf{B}[i]]$. Once again, there may be simultaneous writes at the same location and at least one of them will succeed. Finally, we perform reduction on $\mathbf{F}$ and outcome of the reduction is our desired support value of the DFS extension. The mapping array can be retained for computation of all extensions in $\mathbf{EXT}_k$. We tried several variants of the support computation, namely, 1) *single-ext*: We process one extension at one time (the one we just described); 2) *single-seg*: We compute support for all extensions in $\mathbf{EXT}_k$ at once. For this variant we need to allocate flag array $\mathbf{F}_{\text{lags}}$ of size $|\mathcal{E}| \cdot g$, where $\mathcal{E}$ denotes the extensions extracted from $\mathbf{EXT}_k$; and 3) *multiple-seg*: We perform support computation for extensions in multiple $\mathbf{EXT}_k$ at once. The size of the array $\mathbf{F}$ in this case would be $|\mathcal{E}| \cdot g_{max}$, where $g_{max}$ indicates the maximum mapped graph ids of all segments, $\mathbf{EXT}_k, 1 \leq k \leq |\mathcal{R}(P)|$. To save memory, we limit the size of $\mathbf{F}$ and perform the support computation by parts in multiple $\mathbf{EXT}_k$. The difference, from the algorithmic point of view, between variant 1 and others is just replacement of reductions(scans) by segmented reductions(scans).

In Fig. 5 we show support computation for the extensions obtained from the vertices 0 and 1 of $P$. For each extension, we have three entries in the flag array, $\mathbf{F}$ (since there are $g = 3$ graphs in those segments). Therefore, a 1 indicates the presence of the extension in the corresponding graph in $\mathcal{D}$. We perform a segmented reduction on $\mathbf{F}$ and obtain the support values for the extensions as shown at the bottom of Fig. 5. For $minsup = 2$, the only frequent extension from $P$ is $(1, E)$ in $G_1$ and $G_3$ (see Fig. 2).

## 7. Growing the embeddings

The frequent extensions from pattern $P$ (with $p$ vertices) can obtained using parallel primitives as detailed here for the two types of extensions.

**Algorithm 4** Forward-Ext($\mathbf{EXT}_k$, Extension $e = (v_i, v_j, l_i, l_{i,j}, l_j)$)

```
Summary: construct new column $\mathbf{Q}_{p+1}$ by copying information from $\mathbf{EXT}_k$
Allocate array $\mathbf{M}$ of size $|\mathbf{EXT}_k|$ and fill with 0.
(GPU step) Match elements $e$ in $\mathbf{EXT}_k$ and store 1 in $\mathbf{M}$ on matched positions.
(GPU step) Exclusive scan of $\mathbf{M}$ resulting in index $\mathbf{S}$.
Allocate array $\mathbf{Q}_{p+1}$ that stores the new extensions of the embeddings.
(GPU step) Copy the matched elements $\mathbf{EXT}_k$ into $\mathbf{Q}_{p+1}$ using the index $\mathbf{S}$.
return $\mathbf{Q}$
```

**(1) Forward extensions:** These introduce a new vertex into the pattern. Therefore, we have to construct a new column $\mathbf{Q}_{p+1}$ using information from the $k$-th segment of $\mathbf{EXT}$, i.e., $\mathbf{EXT}_k$. The extraction proceeds as follows, as shown in Algorithm 4: a) Create an array $\mathbf{M}$ of size $|\mathbf{EXT}_k|$ filled with 0s. b) Launch $|\mathbf{EXT}_k|$ threads, where thread $t_i$ stores 1 in $\mathbf{M}[i]$ if $\mathbf{EXT}_k[i]$ matches the DFS code of the extension $e$. c) Perform exclusive scan on $\mathbf{M}$ to create an index $\mathbf{S}$. d) Allocate a new embedding column $\mathbf{Q}_{p+1}$ of size $\mathbf{S}[|\mathbf{EXT}_k|]$, and store the embedding information into $\mathbf{Q}_{p+1}$ at the positions given by $\mathbf{S}$.

**Algorithm 5** Backward-Ext($\mathbf{EXT}_k$, $\mathbf{Q}_p$, Extension $e = (v_i, v_j, l_i, l_{i,j}, l_j)$)

```
Summary: filter $\mathbf{Q}_p$ using information from $\mathbf{EXT}_k$
Allocate array $\mathbf{M}$ of size $|\mathbf{Q}_p|$ filled with 0.
(GPU step) Match $e$ in $\mathbf{EXT}_k$ and store 1 in $\mathbf{M}$ on matched positions given by back-links.
(GPU step) Exclusive scan of $\mathbf{M}$ resulting in indexing $\mathbf{S}$ into new array.
Allocate array $\mathbf{Q}'_p$ that stores the filtered column $\mathbf{Q}_p$.
(GPU step) From $\mathbf{Q}_p$ copy into $\mathbf{Q}'_p$ $i$-th element if $\mathbf{M}[i] = 1$ using the index $\mathbf{S}$.
return $\mathbf{Q}'$
```

**(2) Backward extensions**: These extensions do not introduce a new vertex into the pattern, and thus we filter the last embedding column $\mathbf{Q}_p$ by removing such vertices that do not contain the backward edge, as shown in Algorithm 5: a) allocate a vector $\mathbf{M}$ of the same size of the last column and initialize with 0s; b) execute $|\mathbf{M}|$ threads; thread with index $i$ stores 1 in $\mathbf{M}[i]$ if there exists a back-link from the elements of $\mathbf{EXT}$; and c) perform exclusive scan on $\mathbf{M}$, and store the results in array $\mathbf{S}$. The array $\mathbf{S}$ gives the new positions in the filtered column. Finally, we copy the information from the last column, $\mathbf{Q}_p$ into a new column $\mathbf{Q}'_p$. Thread $t_i$ checks whether $\mathbf{M}[i] = 1$, and if so it copies the element from $\mathbf{Q}_p$ into $\mathbf{Q}'_p$ at position $\mathbf{S}[i]$.

## 8. Experiments

We implemented our GPU-based graph mining algorithm using C++ NVIDIA CUDA library (version 5.5). For parallel primitives, such as inclusive/exclusive scan, reduction and their segmented versions, we augmented the modern GPU library (http://www.moderngpu.com). The sequential gSpan algorithm[1] was executed on a 16 core CPU, the AMD Opteron 6272 "Interlagos" processor (2.1 GHz base clock rate, with up to 2.9GHz in turbo mode), running Ubuntu SMP and having 256GB of memory and 2MB of cache. The GPU implementation was run on a Tesla C2075 GPU, which consists of 448 cores, 6GB of memory and supports CUDA compute capability 2.0. We could not compare with the GPU implementation in (Wang et al., 2013), since the authors did not respond to our request for the code. Further, they report that the sequential gSpan with 35% minimum support took 70.7 seconds on a 3.0Ghz (3.3GHz turbo) Intel Core i5-2320 processor, which suggests a very inefficient sequential/CPU implementation, since our sequential gSpan on the 2.1Ghz (2.9Ghz turbo) AMD Opteron 6272 processor takes less than 4 seconds for the same settings. This suggests that their reported speedups may not be reliable, but rather are likely to be overly optimistic.

### 8.1. Datasets

We use four real-world and two synthetic datasets. The *Protein* dataset was created using 6875 proteins from the RCSB Protein Data Bank[2]. The *NCI* dataset consists of 25595 chemical compounds[3]. The *Citation* dataset consists of citations in high-energy physics journals from SNAP[4]. A graph was created for each available month (from 1993 to 2003) for the top 15 frequently citepd journals. The number of citations for each node was restricted to 10. The *DBLP* dataset was created from the `dblp.xml`[5] entries for 21 years (from 1990 to 2010) considering conference proceedings in 26 different areas.[6] The number of authors was restricted to at most 5, and each pair of authors have to collaborate at least twice in order to have an edge between them. The labels for authors are chosen to be their active area of research in that year, and the largest 20 components were considered for each year. Two synthetic graph datasets were generated using FSG generator (Kuramochi and Karypis, 2001). For generating these datasets the average graph size and the average pattern size were chosen as 30 and 4, respectively.

Some properties of these datasets are listed in Table 1. *Protein* has relatively large sized graphs, whereas the others are smaller. It has a high average degree and relatively high clustering coefficient. *NCI* is the largest among all real-world datasets. However, the graphs are small with a moderate average degree (avg. deg) and low clustering coefficient

[1] Taku Kudo's code www.nowozin.net/sebastian/gboost.

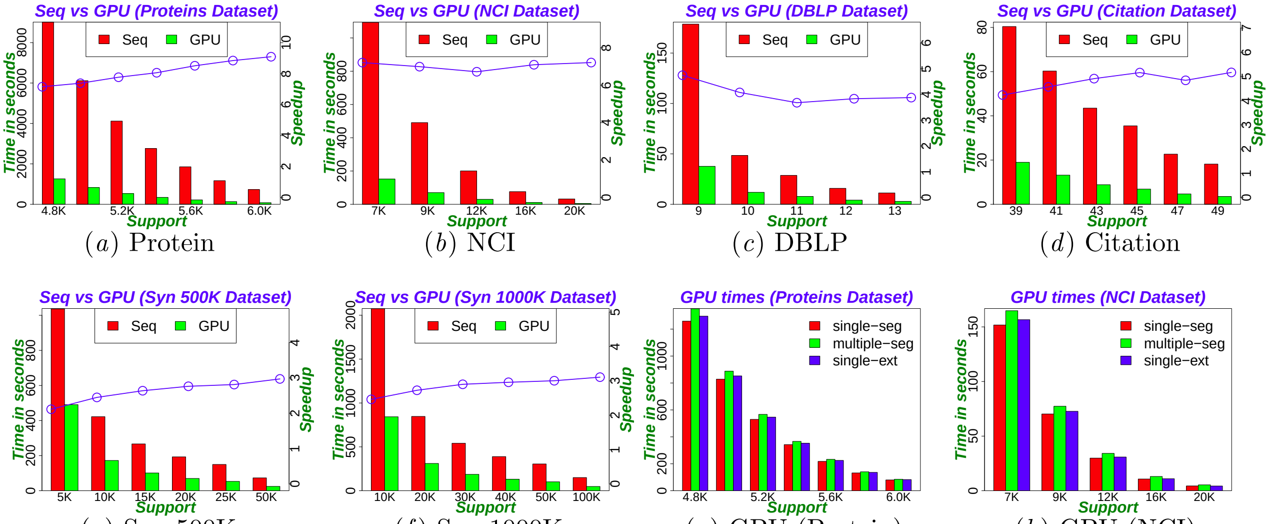
[2] http://www.rcsb.org/pdb

[3] http://cactus.nci.nih.gov/download/nci

[4] http://snap.stanford.edu/data/cit-HepTh.html

[5] http://www.informatik.uni-trier.de/~ley/db

[6] http://en.wikipedia.org/wiki/List_of_computer_science_conferences



(*a*) Protein (*b*) NCI (*c*) DBLP (*d*) Citation

(*e*) Syn 500K (*f*) Syn 1000K (*g*) GPU (Protein) (*h*) GPU (NCI)

Figure 6: GPU versus Sequential Performance

| dataset | $\vert\mathcal{D}\vert$ | avg. $\vert V\vert$ | avg. $\vert E\vert$ | $V$ lbls | $E$ lbls | avg. deg | avg. clustering coeff. |
|---|---|---|---|---|---|---|---|
| Citation | 81 | 469 | 541 | 15 | 1 | 2.042 | 0.008 |
| Protein | 6875 | 1113 | 4392 | 22 | 1 | 7.891 | 0.554 |
| DBLP | 21 | 205 | 300 | 26 | 26 | 2.796 | 0.594 |
| NCI | 24595 | 39 | 78 | 54 | 1 | 3.954 | 0.0014 |
| Syn500K | 500K | 25 | 31 | 5 | 5 | 2.517 | 0.383 |
| Syn1000K | 1000K | 25 | 31 | 5 | 5 | 2.517 | 0.383 |

Table 1: Properties of the datasets

(avg. CC). *Citation* has a very low clustering coefficient as triangles are practically non-existent. *DBLP* contains a large number of cliques (we observed many 5-cliques). This is evident due to the high clustering coefficient. The two synthetic datasets have identical properties.

## 8.2. Performance Results

We report in Fig. 6 the major results from our experiments. The bar plots show the performance comparison of sequential and GPU-based graph mining on the different datasets with various support values displayed on the x-axis. The y-axis on the left shows the time in seconds for the bars, and the y-axis on the right indicates the GPU speedup compared to the sequential CPU version. For the GPU implementation in “Seq vs GPU” plots we consider the ‘single-seg’ approach from Section 6, since it demonstrates the best overall performance among all versions. [7]

The results show that we achieve a consistent speedup of 8 to 9 on *Protein* over the sequential CPU run. For *NCI* the speedup was around 7. However, for *DBLP* and *Citation* we observe a consistent speedup of around 4 and 5, respectively. In general, for denser graphs, such as *Protein*, we observe better speedups. The *DBLP* dataset has many larger cliques, resulting in a large number of embeddings which may cause an increase in the

[7] Note that the speedups reported are with all 448 GPU cores. Since the Tesla GPU threads run in warps of 32, and blocks of threads are scheduled automatically by the GPU on the symmetric multiprocessors, it is not very meaningful to show traditional speedups charts that vary by the number of cores.

| level | Protein | NCI | DBLP | Citation | Syn500K | Syn1000K |
|---|---|---|---|---|---|---|
| 1 | 121379 | 208010 | 1041 | 3536 | 45871 | 91668 |
| 2 | 94000 | 131287 | 2344 | 3878 | 19336 | 38652 |
| 3 | 117581 | 151537 | 8845 | 7686 | 14649 | 29268 |
| 4 | 157530 | 204052 | 22112 | 20935 | 13151 | 26297 |
| 5 | 157663 | 215258 | 49873 | 69061 | 12273 | 24503 |
| 6 | - | 198693 | 103170 | 154183 | 10447 | 20814 |
| 7 | - | 269869 | 100692 | 351918 | 9523 | 18976 |
| 8 | - | 297180 | 120243 | 106500 | 8840 | 17487 |

Table 2: Average number of embeddings per level (upto level 8).

| dataset | supp | #pat | avg pat size | max pat size | avg #bkwd edges | max #bkwd edges |
|---|---|---|---|---|---|---|
| Citation | 39 | 630 | 3.83 | 8 | 0 | 0 |
| Protein | 4.8K | 9436 | 3.00 | 5 | 0.07 | 1 |
| DBLP | 9 | 1048 | 7.16 | 11 | 1.19 | 4 |
| NCI | 7K | 1591 | 8.37 | 13 | 0.49 | 1 |
| Syn500K | 5K | 45470 | 4.54 | 11 | 0.21 | 4 |
| Syn1000K | 10K | 45757 | 4.55 | 11 | 0.21 | 4 |

Table 3: Pattern Properties with Lowest Support

speedup with lower *minsup*. Finally, on the synthetic datasets, due to the sparse graphs, we obtain lower speedups of around 3.

We also show, in the barplots titled ‘GPU times’, the performance comparison among the three GPU versions we implemented, namely ‘single-ext’, ‘single-seg’ and ‘multi-seg’, which were described in Section 6 for parallel support computation of the extensions. We observed that the version ’single-seg’ performs the best among the three. However, we did not observe significant difference in speedups among the versions. This may be due to the fact that the large array for support computation incurs huge I/O overhead. Therefore, even though we process multiple segments in parallel with ‘multi-seg’ variant we do not observe better speedup.

As seen above, we observe different speedups with different datasets. In general the more the number of embeddings, the higher the speedup, as shown in Table 2, e.g., *Protein* and *NCI* have higher, whereas the synthetic ones have lower speedups. Table 3 shows some statistics of the obtained patterns using the lowest minimum support value. The patterns in *Citation* do not have any backward edges as they are mostly star graphs. *Proteins* has smaller patterns. On the other hand, *DBLP*, *NCI* and the synthetic datasets have larger patterns. *DBLP* contains a high number of cycled graphs. The synthetic datasets yield many patterns, but the average number of backward edges is not very high.

## 9. Discussion and Conclusions

GPUs have become attractive for general purpose parallel computing. However, programming them is quite different from the traditional multi-threaded or multi-core programming paradigm. We investigate in detail the challenges of implementing frequent graph mining on GPUs. Our implementation achieves speedups up to 9 for real-world datasets and around 3 for synthetic datasets. We used datasets with various characteristics to examine the effectiveness of our GPU-based algorithm.

We observed that GPU memory allocation and I/O transfers between the CPU and the GPU incur a large overhead. This affects the speedup of our GPU-based algorithm. However, the memory transfers are necessary for fetching extensions and computing supports, etc. Memory allocation of various arrays during the computation is also unavoidable. Therefore, we minimized as many calls to `cudaMalloc` and `cudaMemcpy` as possible. This was mainly done by statically allocating the memory blocks for all the parallel operations, and resizing them when necessary. The most important parallel operations for our algorithms are: 1) parallel scan; 2) parallel *segmented* scan; 3) parallel reduction; 4) parallel *segmented* reduction. These operations also influence the speedup of the GPU implementation as they need a lot of sequential administration. We implemented efficient parallel primitives by studying the Modern GPU documentation, and showed that this indeed makes a substantial difference.

In future, we will consider alternative approaches for GPU-based implementation. Our work can be extended to other graph mining tasks, such as closed graph, maximal graph and temporal graph mining on GPUs. Our current algorithm runs on NVIDIA Fermi Architecture that uses CUDA compute capability 2.0. We plan to explore advanced features of NVIDIA's new Kepler architecture, such as the dynamic parallelism. We also intend to study efficacy of GPU-based parallelism versus multi-core CPUs and other accelerators, such as Intel's Many Integrated Core Xeon Phi coprocessor (61 cores, 1.238Ghz clock rate), for similar tasks.

## References

Gregory Buehrer, Srinivasan Parthasarathy, and Yen-Kuang Chen. Adaptive parallel graph mining for cmp architectures. In *IEEE ICDM Conference*, 2006.

Vineet Chaoji, Mohammad Al Hasan, Saeed Salem, and Mohammed J Zaki. An integrated, generic approach to pattern mining: data mining template library. *Data Mining and Knowledge Discovery*, 17(3):457–495, 2008.

Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artificial Intelligence Research*, 1(1):231–255, 1994.

Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *IEEE ICDM Conference*, 2003.

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD Conference*, 2000.

M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *IEEE ICDM Conference*, 2001.

Wenqing Lin, Xiaokui Xiao, and Gabriel Ghinita. Large-scale frequent subgraph mining in mapreduce. In *IEEE ICDE Conference*, 2014.

S. Nijssen and J.N. Kok. A quickstart in frequent structure mining can make a difference. In *ACM SIGKDD conference*, 2004.

J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

Fei Wang, Jianqiang Dong, and Bo Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In *IDEAL Conference*, 2013.

Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *IEEE ICDM Conference*, 2002.