Anti-Section Transitive Closure

Conference Paper · December 2021

DOI: 10.1109/HiPC53243.2021.00033

6 authors, including:

Oded Green
NVIDIA
48 PUBLICATIONS **771** CITATIONS
SEE PROFILE

David Bader
New Jersey Institute of Technology
407 PUBLICATIONS **7,719** CITATIONS
SEE PROFILE

Anti-Section Transitive Closure

Oded Green* — *NVIDIA*
Zhihui Du* — *New Jersey Institute of Technology*
Sanyamee Patel — *New Jersey Institute of Technology*
Zehui Xie — *Stevens Institute of Technology*
Hang Liu — *Stevens Institute of Technology*
David A. Bader — *New Jersey Institute of Technology*

***Abstract*—The transitive closure of a graph is a new graph where every vertex is directly connected to all vertices to which it had a path in the original graph. Transitive closures are useful for reachability and relationship querying. Finding the transitive closure can be computationally expensive and requires a large memory footprint as the output is typically larger than the input. Some of the original research on transitive closures assumed that graphs were dense and used dense adjacency matrices. We have since learned that many real-world networks are extremely sparse, and the existing methods do not scale. In this work, we introduce a new algorithm called Anti-section Transitive Closure (ATC) for finding the transitive closure of a graph. We present a new parallel edges operation – anti-sections – for finding new edges to reachable vertices. ATC scales to massively multi-threaded systems such as NVIDIA's GPU with tens of thousands of threads. We show that the anti-section operation shares some traits with the triangle counting intersection operation in graph analysis. Lastly, we view the transitive closure problem as a dynamic graph problem requiring edge insertions. By doing this, our memory footprint is smaller. We also show a method for creating the batches in parallel using two different techniques: dual-round and hash. Using these techniques and the Hornet dynamic graph data structure, we show our new algorithm on an NVIDIA Titan V GPU. We compare with other packages such as NetworkX, SEI-GBTL, SuiteSparse, and cuSparse.**

***Index Terms*—Transitive closure, dynamic graph, GPU, parallel graph algorithm**

I. Introduction

Graphs are a natural way to represent real-world problems, found in numerous domains such as social sciences, biological systems, and information systems (to name just a few). Formulating problems with graphs enables using a wide range of well-researched and established computational tools. The GraphBLAS [1] is one such effort to create a framework for implementing graph algorithms through the language of linear algebra.

Theoretical work dating back half a century focused on highly connected graphs, commonly referred to as dense networks. In these networks, most vertices have a direct connection to the remaining vertices in the graph. We have since learned that many real-world networks are, in fact, sparse where each vertex has a small fraction of its potential connections. We commonly refer to such networks as sparse networks. Many sparse networks, social networks in particular, have an additional property - their edge distribution follows a power-law distribution [2]–[4].

* These authors contributed equally to the manuscript.

A transitive closure [5] is a basic binary relation operator for finding reachability between two entities in a graph. Given a graph G, if (u, v) and (v, w) are two different edges in G, the edge (u, w) should appear in the output of the transitive closure G_{TC}. The transitive closure allows for efficient querying if two vertices share a directed path. To be precise, such querying is possible with $\mathcal{O}(1)$ operations. Applications requiring a large number of queries benefit from the transitive closure operation.

Transitive closures are used in numerous domains, including: database query languages [6], [7], VLSI Test Generation [8], detecting runtime deadlocks [9], compiling optimization [10] such as redundant synchronization removal and the legality of iteration reordering transformations testing [11]. Transitive closures are also used for model checking [12], RNA secondary structure prediction [13], and machine learning [14]. Demetrescu and Italiano [15] show that small changes to the transitive closure method can generate the shortest path. The list of applications using transitive closure continuously grows, and these entail a more efficient transitive closure implementation method with high practical performance.

This paper introduces a new algorithm for finding the transitive closure of a graph called Anti-section Transitive Closure (ATC). The major algorithmic contributions are as follows.

- We introduce a novel *list anti-section* operator that shares many algorithmic traits with a set intersection. The sorted set intersections found in a breadth of triangle counting algorithms are well understood and optimized. Thus, our algorithm benefits from them as well.
- The simplicity of the anti-section operator allows for effective parallelization across tens of thousands of threads. The overlap between the two operators and their simplicity leads to high productivity and enables us to quickly implement this algorithm for NVIDIA GPUs.
- The proposed ATC algorithm integrates with a dynamic graph data structure which supports bulk edge insertions for updating the intermediate results of the transitive closure. By using a dynamic graph data structure, we need to store a single graph in memory resulting in a reduced memory. Further, the algorithm is simple as it does not require complex locking mechanisms to update the graph. Altogether, these algorithmic contributions allow for putting together a simple, elegant, and scalable

solution for transitive closures.

Based on the proposed ATC algorithm, we have the following performance results.

- We use Hornet, a dynamic graph data structure, for inserting edges into the output graph. Using Hornet, we show a reduced memory footprint compared to other methods that require rebuilding the graph when new edges are found.
- Our new hash-based ATC algorithm outperforms both single-thread CPU, multi-thread CPU, and GPU-based implementations. Our hash-based ATC can resolve the problem of identifying duplicated edges with an acceptable memory overhead while improving performance over our dual-round algorithm and other algorithms.
- Our GPU-based implementation outperforms a different GPU implementation by $10\times$ and $100\times$ for two inputs on the same GPU. When comparing our algorithms against NetworkX, GraphBLAS-GBTL, and GraphBLAS-SuiteSparse, the speedup varies depending on the number of threads, programming language, and problem formulation. Compared to the parallel and highly efficient GraphBLAS-SuiteSparse framework, our algorithm can be upto $3\times$ faster.

II. Problem Formulation

The transitive closure of a graph is defined as follows. Given a directed graph $G =< V, E >$ where V is the set of vertices and E is the set of edges, the transitive closure

$$G_{TC} =< V, E_{TC} >$$

of G has the same vertices as V and $E \subseteq E_{TC}$. If $\forall e \in E, e \in E_{TC}$ and if $e_1 = (u, w) \in E_{TC} \wedge e_2 = (w, v) \in E_{TC}$ then $e = (u, v) \in E_{TC}$. The transitive closure graph can directly answer the reachability query between any vertex pair in $\mathcal{O}(1)$ time. The output of the transitive closure operation on an input graph is a new graph where reachable vertices are directly connected. Transitive closures are especially interesting for directed graphs. Using a connected component algorithm for an undirected graph can produce the same result and will be quicker than most transitive closure algorithms, and will not require the increased memory footprint.

Many transitive closure algorithms work iteratively. These algorithms continuously add new edges into the graph as these are found. The challenge of scaling these approaches is non-trivial, primarily due to their dependency on dynamic graph data structures. Implementing a high-performance dynamic graph data structure requires tackling several problems: managing fine-grain thread control, which leads to deadlocks or livelocks, memory allocation and memory management, data placement, and much more. For this reason, transitive closure implementations for sparse graphs will use multiple graph instances in their implementations, resulting in an increased memory footprint.

We show a transitive closure algorithm designed for sparse graphs that uses a dynamic graph data structure. Our implementation uses the Hornet [16] dynamic graph data-structure. Hornet shares many traits with the Compressed Sparse Row (CSR) format. Unlike CSR, Hornet supports both edge insertions and deletions. Hornet supports efficient and scalable update operations for large and sparse networks. Using Hornet, we require storing only a single graph in memory throughout our execution.

The kernel operation to generate the new edges of a transitive closure is called anti-section in this paper. The term anti-section stems from the execution similarity to a set intersection operation. Anti-sections focus on finding non-common elements in one of the sets versus finding the common elements. Given a directed edge $(u, v) \in E_{TC}$, let $adj(u)$ be the set of u's adjacent vertices and $adj(v)$ be the set of v's adjacent vertices , mathematically the anti-section of an edge (u, v) can be defined based on a *set difference* as follows:

$$AntiSection((u, v)) = \{(u, p) | p \in adj(v) - adj(u))$$

For any edge $e \in AntiSection((u, v))$, that edge should be identified and added to the output graph. In the following section, we will describe how we implement the $anti-section$ operation efficiently.

III. Proposed Method

Our anti-section transitive closure generation method includes three key components: anti-section new edges generation, dynamic graph bulk edges insertion, and dynamic graph segmented sorting.

A. Anti-Section for New Edges Generation

Given a single edge, the anti-section operation is the kernel operation in which new edges are generated. These edges are added one at a time in a batch. Lastly, the batch is inserted into the graph in a bulk fashion. Furthermore, tens of thousands of such operations are executed concurrently. In Alg. 1, the anti-section operation shares a large number of traits with a sorted-list intersection operation. For the sake of simplicity, the pseudo-code algorithm assumes that the set add operation is feasible at scale [1].

In Alg. 1, we first advance a_i to keep $A[a_i] \geq B[b_i]$ or until a_i points to the last element because we want to check if $B[b_i]$ is in A. So, we can safely skip all $A[a_i]$ adjacency vertices whose ID value is smaller than $B[b_i]$ as they do not require comparing with the current elements from b_i to the end in B. When we find an element $B[b_i]$ which is not in A (meaning that $B[b_i]$ is larger or less than any elements in A or is sitting between two neighbour elements such as $A[a_{i-1}]$ and $A[a_i]$ of the ordered array A), a new edge $(u, B[b_i])$ will be inserted into the dynamic graph G.

At runtime, for a given edge (u, v), our scheduler can partition one complete anti-section operation into several parallel sub anti-section operations by dividing $adj(v)$ into subsets. In this way, we can achieve two effects: (1) more fine-grain parallelism operations and (2) better load-balancing of the kernel anti-section operation.

[1]On modern GPU architectures placing elements at the end of an array has good performance due to the low overhead of atomic instructions

Algorithm 1: Anti-Section Algorithm

```
1  AntiSection((u,v), A, B)
   /* (u,v) is the given edge. A = adj(u) and
      B = adj(v) or B is a subset of adj(v)          */
2  a_i ← 0; b_i ← 0; count ← 0; NewEdgeSet = φ
3  while (b_i < |B|) do
4      while (A[a_i] < B[b_i] and a_i < |A| − 1)) do
5          a_i ← a_i + 1;
6      end
7      if (A[a_i] == B[b_i]) then
8          b_i ← b_i + 1;
9          continue;
10         if (B[b_i] ≠ u) then
11             NewEdgeSet.add((u, B[b_i]))
12             b_i ← b_i + 1;
13             count ← count + 1;
14         end
15     end
16 end
17 return NewEdgeSet
```

Algorithm 2: Hash Search based Anti-Section Algorithm

```
1  AntiSection((u,v), A, B)
   /* (u,v) is the given edge. A = adj(u) and
      B = adj(v) or B is a subset of adj(v)          */
2  a_i ← 0; b_i ← 0; count ← 0
3  while (b_i < |B|) do
4      while (A[a_i] < B[b_i] and a_i < |A| − 1) do
5          a_i ← a_i + 1;
6      end
7      if (A[a_i] = B[b_i]) then
8          b_i ← b_i + 1; continue;
9      end
10     if (B[b_i] ≠ u) then
11         insertSuccess = hashTable.insert({u, B[b_i]})
12         if (insertSuccess == true) then
13             NewEdgeSet.add((u, B[b_i]))
14         end
15         b_i ← b_i + 1;
16         count ← count + 1;
17     end
18 end
19 return NewEdgeSet;
```

Our anti-section operation and the intersection operation in triangle counting share some traits: 1) both use a sorted adjacency list, 2) both can execute on different edges in parallel, and 3) both have the same time complexity. The big difference between these two operations is the values that they search. An intersection looks for common values; whereas, the anti-section operation looks for values in v's adjacency list that are non-existent in u's adjacency list. The new edges generated from different anti-section operations can be merged together and added into a batch operation. One of the key benefits of our anti-section operator being so close to the sorted-list intersection operator is that it is used in many graph triangle counting algorithms. The last decade has a plethora of optimizations for triangle counting (Sec. VI-B) such as vertices partition, merge-path, binary-search and hash method.

B. Bulk based New Edge Set Creation and Insertion

We describe two approaches for managing and storing the newly found edges in our anti-section transitive closure operation. These two approaches trade off storage complexity and time complexity. We name these two methods as the "dual-round" approach and the "hash-based" approach.

Dual-Round: The dual-round method requires two iterations of the anti-section operation. The first iteration counts the number of newly detected edges without storing them for allocating the exact array size for storing all the new edges – this is the batch array. In the second iteration of the anti-section, we place the newly found edges into the batch array. Alg. 1 does not change a lot between the two rounds.

Different anti-section operations operating on different adjacency lists can find the same edge, resulting in a duplicate edge in the batch array. When the number of duplicated edges is low, the storage complexity of this approach is also low. The storage requirements increase with the number of duplicates. From a theoretical perspective, this does not increase the time complexity of the algorithm. In practice, the runtime does increase. The simplicity of this approach enables us to give a tight complexity bound (Sec. IV-B).

Lastly, the duplicate edges require a filtering phase to clean the data to avoid inserting duplicates [2]. For brevity, we do not cover this topic in considerable detail as it can span numerous papers. Instead, we refer the reader to Sec. VI-C.

Hash: In practice, we found that for many networks, the number of duplicate edges found was extremely high and created a strain on the amount of memory needed to store the new edges. To avoid the problem of edge duplication, we use a simple hash-table like data structure. Our hash table is a simple array of length H where each entry stores only "0" and "1" values to state if an entry is in use or not. Given a newly found edge $e = (u, v)$, we hash the edge with $h(e)$ and check the value at $table[h(e)]$. If $table[h(e)] == 0$, then this is an edge that should be added to the batch. If $table[h(e)] == 1$ then one of two things has happened: either the edge e has already been added into the table or a different edge, $\hat{e}$, was added into the table (false-positive). Either way, we will **not** attempt to find an empty entry in the hash-table and will not insert the edge to the batch. Specifically, if $table[h(e)] == 1$ and the edge that caused this value to be set to 1 was the edge e then all is fine. However, if $table[h(e)] == 1$ was triggered by edge $\hat{e}$, then it means that the edge e will not get inserted in this iteration. However, the way that this algorithm operates, this edge is found in the following iteration. Specifically, transitive closure algorithms do not stop until there is an iteration with zero new edges found. Since this edge needs to be added and that the current iteration has at least one found edge, we can be certain that a follow-up iteration will find the edge as part of the continued progress. The hash table is cleared in each iteration of transitive closure. Pseudo-code for the hash-table insertion is available in Alg. 2. **Thus, it should be clear that the hash-based anti-section operation not only**

[2]The data structure, Hornet [16], that we use for implementing our new algorithm supports this type of filtering. We bring this point up as it creates many algorithmic challenges and increases the computational requirements if not dealt with properly.

finds the necessary edges but it is also responsible for the deduplication process.

Trade-off of Dual-Round and Hash: The hash-based approach has the added benefit that it avoids generating duplicate edges. However, it comes at the cost of its increased theoritical storage complexity of $\mathcal{O}(H)$, where H is the memory size for storing the hash table. In practice, this additional memory is relatively small compared to the amount needed for the duplicated edges. Finding a good size for H compromises the amount of storage size and the number of iterations. Allocating a small hash-table will result in the table filling up quickly and needing more iterations. Thus, we cannot give the same tight bound on the number of iterations required by the algorithm for completion. However, the hash-based approach reduces the amount of anti-section operations compared to the dual-round method. Further, the hash-based approach does not require filtering as no duplicate edges exist in the batch. In practice, the hash-based approach outperforms the dual-round approach.

Each anti-section operates on a different directed edge. We can use the result of the anti-section to generate a set of edges, possibly the null group, which needs to be inserted into the graph. Our algorithm will run $|E^i_{TC}|$ anti-sections in each iteration where E^i_{TC} is the set of edges in the graph by iteration i. As the insertion works in a bulk synchronous manner, newly detected edges are not inserted until all anti-section operations have been completed. Instead, edges get inserted into the graph in a bulk fashion[3]. For dual-approach, duplicate edges are filtered out in a post-processing phase to avoid inserting the same edge numerous times.

An asynchronous anti-section algorithm might be feasible; however, such an algorithm will face many performance problems. As the anti-section operation requires sorted adjacency lists, an asynchronous algorithm must ensure that the inserted edge is placed correctly in its new adjacency array. Otherwise, other anti-sections could be impacted by possibly iterating over a non-sorted array. The problem is exacerbated by the increase in the number of threads found in modern systems. For this reason, we chose the bulk synchronous approach.

C. Dynamic Graph Based Segmented Sorting

The Hornet data-structure [16] is a framework designed to solve both static and dynamic graph problems. Hornet is also an excellent fit for problems where the graph evolves as part of the algorithm — precisely as required by our new formulation. Rather than building a new transitive closure graph in each iteration, we continuously add edges into the graph until the operation is complete. Hornet supports both a static graph back-end, similar to CSR, and a dynamic graph back-end. As our algorithm requires edge insertions, we use the dynamic graph back-end. Since the different transitive closure iterations need the graph to be sorted, this is a requirement expected of Hornet. Hornet uses an efficient segmented-sorting algorithm based on logarithmic radix binning [20] after each bulk edge insertion to ensure that the graph is sorted for the next iteration of the algorithm.

[3]Bulk edge insertion is a key feature of several existing dynamic graph data structures, including Hornet [16], cuSTINGER [17], STINGER [18], and AIMS [19]

Algorithm 3: Dynamic Graph Iteration based Transitive Closure Generation

```
1  while (true) do
2      globalBatch ← φ
3      forall (u, v) ∈ E_TC do
4          batchUV ← {e|e ∈
             AntiSection((u, v), adj(u), adj(v))}
5          if (batchUV ≠ φ) then
6              globalBatch.append(batchUV)
7          end
8      end
9      if globalBatch == φ then
10         break;
11     end
12     Graph.insert(globalBatch)
13     Graph.Sort()
14 end
```

In a segmented sort, each adjacency list is sorted separately. Segmented sorting tends to be computationally more efficient than sorting the entire graph. Furthermore, the way that the data is stored both in a CSR representation and in Hornet requires transforming the data to edge-pairs prior to its sorting - this increases the memory footprint and requires extra time. Our algorithm uses the segmented-sort algorithm presented by Fox *et al.* [20].

While sorting can be fairly costly for some applications, its cost is not considerable compared to the anti-section operations. Sorting an adjacency array of size d has a time complexity of $d \cdot \log(d)$ or $\mathcal{O}(d)$ depending on the sorting algorithm used (merge-sort and radix-sort, respectively). In contrasts, the time complexity of the anti-section operation for the same vertex will require roughly $\mathcal{O}(d^2)$ operations (see subsection IV-B). As such, the sort's overhead is not high compared to the anti-section itself and is not a performance bottleneck.

Simplistic pseudo-code for all the phases of ATC is available in Alg.3, which consists of the following: scalable anti-section new edges search (with edge batch creation), bulk edge insertion, and dynamic graph segmented sorting. The pseudo-code does not include the parallel code needed for managing the edge insertion. We discuss the complexities of the insertion in the subsequent subsections. Still, we note that storing the newly found edges of a given anti-section, denoted as *batchUV* in the pseudo-code, is a crucial feature of this algorithm.

IV. Complexity Analysis

In this section, we discuss the iteration bounds, time and space complexities of the new method.

A. Bound of Iteration Times

We prove that our iteration based formulation of ATC converges quickly as it requires at most $\lceil \log_2 |V| \rceil$ times of iteration to completely build the transitive closure G_{TC}.

Lemma IV.1 (Diameter Shrinkage)**.** *The diameter of the graph shrinks by half in each iteration of ATC. $\forall u, v \in V$, if a path*

between u and v exists, then let l be the length of the shortest path connecting them. If $l > 1$, then after the anti-section the shortest path will be denoted with $\hat{l}$ and $\hat{l} \leq \lceil \frac{l}{2} \rceil$.

Proof. Let $(u = p_0, p_1, ..., p_l = v)$ be the shortest path between u and v. Based on the definition of anti-section transitive closure, after one anti-section operation across all the edges in the graph, the following edges will be identified as edges that need to be inserted into the transitive closure graph: $(p_0, p_2), (p_1, p_3), (p_2, p_4)..., (p_{l-2}, p_l)$. If l is an odd number, the following new shortest path between u and v exists in the graph $(p_0, p_2, p_4, ..., p_{l-1}, p_l)$ and its length is $\frac{l+1}{2} = \lceil \frac{l}{2} \rceil$. If l is an even, the new shortest path between u and v is $(p_0, p_2, p_4, ..., p_{l-2}, p_l)$ and its length is $\frac{l}{2} = \lceil \frac{l}{2} \rceil$. Thus, these shortest paths exists and their lengths meet the requirement of $\hat{l}$. □

Lemma IV.2 (ATC Ending Condition)**.** *When the graph's diameter has shrunk to 1, then the transitive closure has been successfully built.*

Proof. Assume by contradiction that there exists a shortest path $path = (p_0, p_1, ..., p_l)$ that has a length, $l > 1$ and that the transitive algorithm has completed. The anti-section of p_0 with p_1 will find p_2 which will result in a new edge being added, in contradiction with the assumption. Therefore, the algorithm will not complete until the diameter is of length 1. □

Theorem IV.3 (Maximum Iteration Steps)**.** *The maximal number of iterations of ATC will be no more than* $\lceil \log_2 |V| \rceil$

Proof. $\forall u, v \in V$, if there exists one path between u and v, then the longest shorted path between u and v in any graph G should not be larger than $|V| - 1$. Based on Lemma IV.1, it will take at most $\lceil \log_2 |V| \rceil$ iteration to shrink the diameter of G to 1. Based on Lemma IV.2, the diameter of G becomes 1 means that the transitive closure has been successfully built, and the algorithm will end. So, at most $\lceil \log_2 |V| \rceil$ iteration will be needed for ATC. □

B. Time Complexity Analysis

For the time complexity analysis, we use the basic anti-section algorithm as described in the pseudo-code of Alg.1. Given the edge $e = (u, v)$, an anti-section operation requires $\mathcal{O}(d_u + d_v)$ operations, where d_x is the degree of vertex x. Given a vertex, u, that vertex will also partake in d_u anti-section operations - one for each of its adjacencies. As a results, the time complexity for a vertex is $\mathcal{O}(d_u^2)$. We use d_{max} to denote the vertex with the largest degree. Since there are V vertices, we can write the time complexity as $\mathcal{O}(|V| \cdot d_{max}^2)$ as $d_{max} \geq d_u$ for all vertices. While this **does not** give a tight complexity for the algorithm, this complexity is identical to that of triangle counting. Many triangle counting algorithms use the above time complexity as no tighter bound is known.

The $\mathcal{O}(|V| \cdot d_{max}^2)$ time complexity is correct for each iteration of our algorithms. The size of d_{max} can increase with the iteration as additional edges are added into the graph. We can rewrite the time complexity per-vertex as $\mathcal{O}((d_u^i)^2)$ where i is the iteration and the time complexity iteration as $\mathcal{O}(|V| \cdot (d_{max}^i)^2)$ The time complexity of transitive closure for dense matrices can be bound by $O(|V|^3 \cdot log(|V|)$. To the best of our knowledge, no tighter bound can be given to SpGEMM as it is a data-dependent problem. Thus, the GraphBLAS solutions do not have a tighter bound than our new algorithm. Fortunately, both algorithms have an equal number of iterations.

For the dual-round approach, we can bound the time complexity of the entire algorithm to be:

$$\mathcal{O}(\sum_{i=0}^{\log_2(|V|)} |V| \cdot (d_{max}^i)^2)$$

C. Memory Complexity Analysis

The storage complexity for transitive closure is non-trivial and data-dependent; however, we can show that our solution has a smaller memory footprint than the linear algebra-based approach. For the linear algebra solution, three graphs are necessary for the matrix multiplication and the memory should be $\mathcal{O}(3 \times |V|^2) = \mathcal{O}(|V|^2)$. In contrast, our solution uses a single graph, and all operations are done in-place on that one graph and the memory should be $\mathcal{O}(|V|) + \mathcal{O}(|E_{TC}|) = \mathcal{O}(|E_{TC}|)$ since $|V| \leq c \times |E_{TC}|$, where c is a constant. For sparse graph, $\mathcal{O}(|E_{TC}|) << \mathcal{O}(|V|^2)$. Therefore, our memory footprint is much smaller than linear algebra-based solutions.

V. Experimental Results

A. Experimental Setup

System and Configuration: Our implementation of Anti-section Transitive Closure is in CUDA and targets NVIDIA GPUs. The GPU used in our experiments is the NVIDIA Titan V from the Volta micro-architecture. The Titan V has 80 SMs (streaming multi-processors) and 64 SPs (streaming processors) per SM, for a total of 5120 SPs, commonly referred to as CUDA threads. The Titan V has a total of 12GB of HBM2 memory and 6MB of shared cache between the SMs. Newer GPUs such as the NVIDIA A100 have almost $7\times$ the amount of memory with 80GB, enabling running on significantly large graph instances. All CPU experiments are executed on a dual Intel Xeon E5-2650 v4 Processor running 2.2GHz with 30MB LLC. The CPU has a total of 512GB of memory. This system has 24 cores with 48 threads.

Input Networks: Our tests use a mix of real-world and synthetic data taken from SuiteSparse collection [21]. Table I summarizes the properties of these graphs, including the number of edges in the output graph. We specifically use directed graphs and report the number of directed edges. We avoid using undirected networks as typically the largest connected component can account for $90\% - 95\%$ of the vertices [21]–[24]. Such components would result in a clique, making it less interesting while also requiring substantial memory. Lastly, note that the ratio of the edges in the input graph and the transitive closure graph can range between almost $60\times$ for

delaunay_n17 and over $1,000X$ for cit-HepPh. For this reason, we are unable to test our algorithm for inputs with millions of edges as the transitive closure graph would have an extremely large memory footprint.

TABLE I
NETWORKS USED IN OUR EXPERIMENTS. $|\hat{E}|$ REFERS TO THE NUMBER OF EDGES IN THE TRANSITIVE CLOSURE OF THE GRAPH. DENSITY VALUES GO BETWEEN 0 (EMPTY) AND 1 (FULL).

Graph Name	$\|V\|$	$\|E\|$	$\|\hat{E}\|$	Density $\|\hat{E}\|$
p2p-Gnutella09	8,114	26,013	21,400,336	0.3255
delaunay_n13	8,192	24,547	1,656,270	0.0250
as-22july06	22,963	48,436	1,477,572	0.0029
delaunay_n15	32,768	98,274	4,703,128	0.0044
delaunay_n17	131,072	393,176	23,499,929	0.0014
cit-HepPh	34,546	421,578	485,646,072	0.4069

Additional Frameworks: We compare the performance of our new algorithm with several implementations: NetworkX, SEI-GBTL [25], SuiteSparse [26], and cuSparse [27]. We chose these implementations as they are either open-source or readily available for us. SuiteSparse and cuSparse are also known for having good sparse implementations.

NetworkX is a widely used graph framework amongst data scientists. The NetworkX implementation uses the DFS (depth-first search) formulation. A DFS traversal is initiated from each root, and the time complexity is $\mathcal{O}(|V|^3)$. This approach is inherently sequential.

We also compare against three transitive closure implementations that use sparse matrix-matrix multiplication formulation. The first of these uses the GraphBLAS implementation of SEI, which we refer to as SEI-GBTL. SEI-GBTL uses sparse adjacency matrices as the basic matrix data structure, and it is sequential. The second is based on SuiteSparse and runs in parallel. The third is based on cuSparse and it is a parallel implementation for the GPU. Altogether, these implementations cover different solutions as well as target different architectures. These frameworks do not include a transitive closure. However, using the well-known BLAS formulation, we implement sparse versions using sparse matrix multiplications.

Implementation Details: The implementation of our new Anti-section Transitive Closure algorithm is in C++ and CUDA. Specifically, all parallel code on the GPU is in CUDA. We use the Hornet framework (described below) for our dynamic graph data structure. Hornet is part of NVIDIA's RAPIDS initiative and is part of cuGraph (referred to as cuHornet in the RAPIDS framework). We include two variations of ATC: dual-round and hash-based.

B. Performance Analysis of ATC

1) Breakdown Analysis: Fig. 1 depicts a detailed breakdown of the execution time for both the dual-round and hash-based algorithms. The top row represents the execution breakdown in percentage, and the bottom row depicts the execution time as a function of the iteration. We report the execution times for each algorithm for the three key phases: anti-section, bulk insertions, and graph sorting. The execution breakdown takes into account the total time of an algorithm for the entire iteration. For example, the sorting time is nearly identical for dual-round and hash in a given iteration. However, the relative cost of sorting is higher for the hash-based ATC. The difference is primarily due to the disparity of the anti-section operation between dual-round and hash. Dual-round executes the anti-section twice and requires an additional processing phase for duplicate filtering. That is why we typically see a $4\times \sim 6\times$ difference in the anti-section execution time and a $3\times \sim 5\times$ speedup of the hash-based approach over the dual-round approach.

Another observation is that the absolute time spent on graph sorting increases with the density. At the same time, we see that time spent on the anti-section increases at a higher pace; thus, the relative cost of sorting decreases.

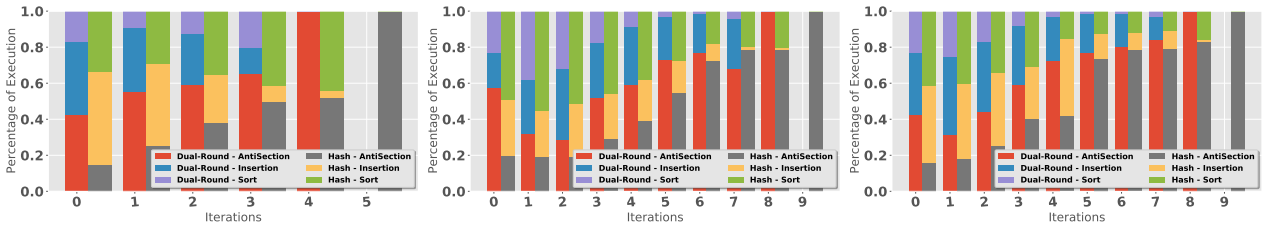
2) Impact of Hash-Table Size: Fig. 2 depicts the impact of the hash-table size on the number of iterations and the execution time of the hash-based transitive closure. With a small hash-table, the algorithm runs additional iterations. In contrast, a large hash-table increases the memory footprint, and the cost of clearing the hash-table at every iteration also increases. Fortunately, clearing the hash-table is not extremely expensive and requires a simple scan across all the hash table entries. The scan uses a bandwidth-friendly memory access pattern. Thus, the reset operation is relatively inexpensive in comparison to the anti-section. The increased memory footprint is more challenging to amortize, so we offer the following rule of thumb: set the hash-table to 10% of the system memory. In the worst case, if the output of the transitive closure is enormous, within ten iterations, the entire system memory will get filled. If the output is not very large, then the hash table will not get used in its entirety. However, fewer iterations of the anti-section will be required.

C. Comparison with Other Frameworks

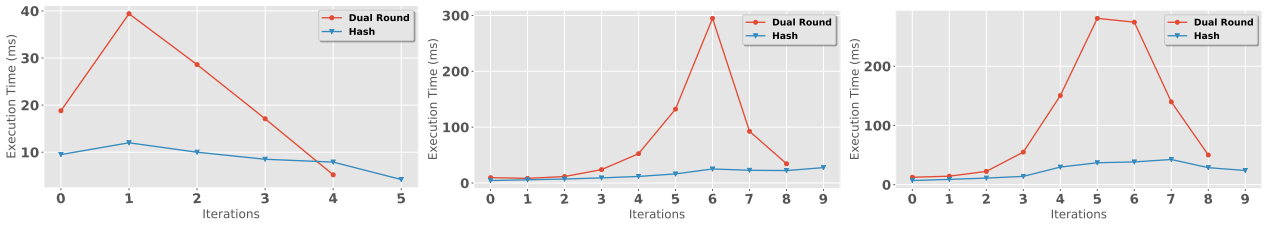
1) Performance Comparison: We start by comparing our new ATC algorithms with the other implementations. Fig. 3 depicts the execution time (left sub-plot) of these implementations, the speedup of ATC dual-round (middle plot), and ATC hash-based (right plot) in comparison to the other algorithms. For execution times, lower is better. For speedups, a value over 1 implies that the algorithm is outperforming the baseline. There were several graphs that we could not collect the execution time as they did not finish due to out-of-memory, segmentation-fault, or timed out. These are denoted with a missing bar.

We found the sequential execution of SuiteSparse was on many occasions $10\times$ faster than SEI-GBTL. Further, SuiteSparse's ability to use multiple threads also improves its performance between $12\times \sim 18\times$ in comparison to its sequential execution. Altogether, SuiteSparse can be $120\times \sim 180\times$ faster than SEI-GBTL.

The hash-based ATC algorithm is usually over $1000\times$ and $400\times$ faster than NetworkX and GraphBLAS SEI-GBTL,

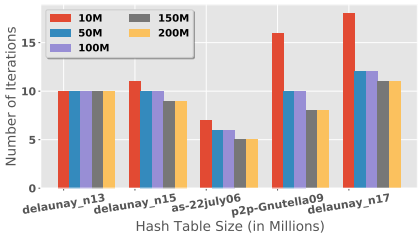


(a) as-22July20 (b) delaunay-n13 (c) delaunay-n15

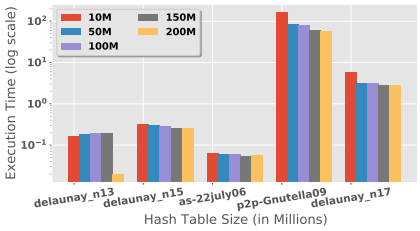


(d) as-22July20 (e) delaunay-n13 (f) delaunay-n15

Fig. 1. The top row depicts the execution breakdown (anti-section, bulk insertions, and graph sorting) in percentage. The execution breakdown is relative to the execution time of a given algorithm. The bottom row depicts the execution time for the same three networks as a function of the iteration.



(a) Iterations as function of hash table size



(b) Execution time (ms) as a function of hash table size

Fig. 2. The hash table size impacts the number of iterations (top) and the execution (bottom). Selecting the ideal hash-table size is important.

respectively. The hash-based ATC typically outperformed cuSparse and SuiteSparse, except for one instance for each of these frameworks.

SEI-GBTL, SuiteSparse, and cuSparse implementations all use matrix-matrix multiplication. Therefore, it is not surprising the GPU's cuSparse can be up to $50\times$ faster than the sequential SEI-GBTL. SuiteSparse's sequential implementation is roughly $10\times$ faster than SEI-GBTL. With its parallel implementation, SuiteSparse is up to $180\times$ than SEI-GBTL. Yet, our new algorithm in most cases is about $2.5\times \sim 3\times$ faster that SuiteSparse. The performance gain is due to both the GPU's compute resources and the different optimization methods of ATC.

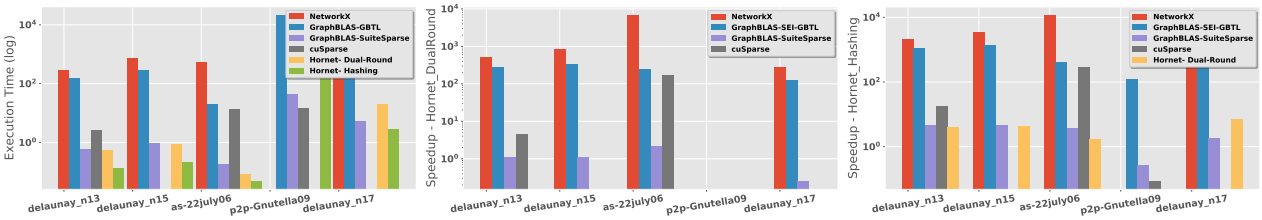
2) Number of Iterations and Per Iteration Analysis: We compare the number of iterations necessary for finding the transitive closure of a graph, Fig. 4. We do not include results for NetworkX as it uses a different (DFS) formulation. All the algorithms require nearly the same number of iterations, with a slight variance. The number of iterations follow our analysis result in the subsection IV-A. The dual-round algorithm always requires fewer iterations than the hash-based approach, which matches the theoretical result in subsection III-B. Note, the increase in the number of iterations is not substantial. In most cases, the increase in iterations pays off as the dual-round algorithm requires additional anti-sections operations.

Lastly, we analyze the execution time as a function of the bulk batch of inserted edges for both our ATC algorithms and cuSparse, Fig. 5. For most iterations, the hash-based approach is over $10\times$ faster than cuSparse, and, in some cases, it is over $100\times$ faster. For the *p2p-Gnutella09* network, there are several iterations where our ATC algorithm is slower. As the graph becomes denser, the cost of the anti-section operation also grows. There seems to be room for improvement for our anti-section algorithm for denser graphs, which we will explore in future work.

VI. Related Work

A. Transitive Closure

Many of the early transitive closure algorithms were designed to operate on dense networks. In particular, the linear algebra approach uses dense matrix multiplication, including the seminal algorithm by Warshall [28]. Warshall's algorithm is also applicable, with some modifications to sparse networks. The research on this topic over the years has been extensive. Warren [29] optimized Warshall's method by processing the matrix elements in a row-wise manner when the matrix is



(a) Execution Time (b) Dual-Round Speedup (c) Hash-Based Speedup

Fig. 3. (Left) - execution time of the various implementations (log-scale). Lower is better. (Middle) and (Right) - speedups of our two anti-section algorithms. Higher is better.

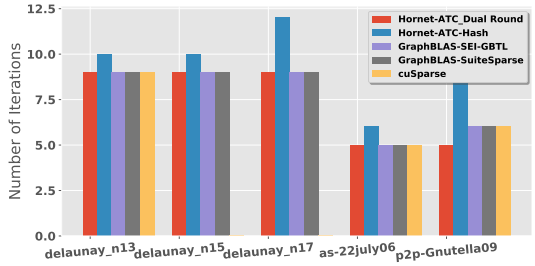


Fig. 4. The number of iterations needed for finding the transitive closure for various inputs.

stored in a row-wise fashion. Pieterse *et al.* [30] shows how different algorithm techniques, such as monitoring for changes, loop fusion, loop tiling, and short-circuiting, can affect transitive closure performance. Hardware-oriented efforts were another research direction. Kung *et al.* [31] developed systolic arrays to find the transitive closure in hardware and Velasquez *et al.* [32] implemented Boolean matrix-based transitive closure algorithm within 3D Crosspoint Memory.

Nuutila [33] developed an optimized dense graph components-based transitive closure algorithm identifying the relationship between strongly connected components of a graph and then generating the transitive closure by utilizing additional information on the graph layout. Some algorithms rely on a predefined processing order rather than depending on early termination, for example, Tarjan's [34] DFS algorithm. Yet, DFS algorithms have proven to be quite challenging to parallelize in the general case. Naumov *et al.* [35] show that DFS can be parallelized with moderate success in the case of a DAG on NVIDIA GPUs; however, the same algorithm is not practical for generic graphs that might include cycles.

The semi-naive algorithm by Bancilhon [36] is an iterative transitive closure algorithm that avoids regenerating existing relations or edges. Our hash bashed method is inspired by this approach. Wang *et al.* [37] use a hash function in their multiprocessor transitive closure algorithm for mapping newly generated tuples to different processors. Gilray *et al.* [38] implemented parallel transitive closure at process level parallelism using MPI. In contrast, our algorithms scale using intra-process parallelism with a large number of lightweight threads.

B. Triangle Counting

Recall, in Sec. III we showed the relationship between our anti-section operation and the intersection operation. However, our **ATC** algorithm differs from a typical triangle counting algorithm. In recent years, we have witnessed a surge of new triangle counting algorithms due to the HPEC GraphChallenge [39]. We also observed exciting efforts such as matrix multiplication-based triangle counting [40], new load-balancing mechanisms [41], [42], and subgraph matching-based triangle counting [43]. We refer the readers to [39] for an extended discussion.

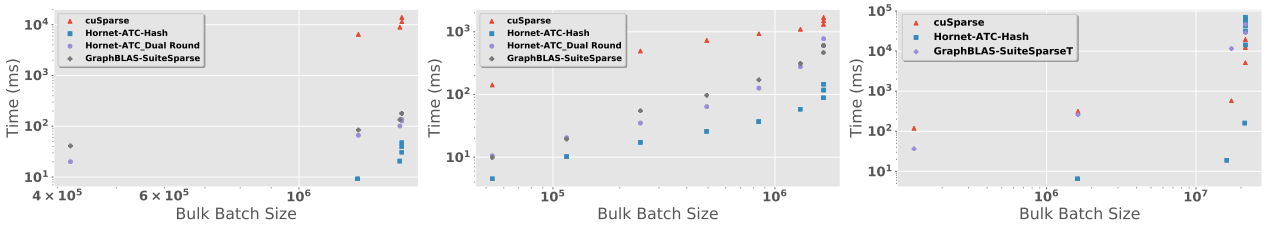
Merge-path based triangle counting. Odeh et al. [44] introduced the Merge-Path concept for parallel merging and sorting. Merge-Path proposes an efficient way to partition the two sorted lists into balanced and disjoint subranges to increase parallelism. Green *et al.* [45] then extended Merge to the GPU. Intersect-Path extends the Merge-Path concept to set intersections operations and triangle counting [46]. In our work, we extend the Merge-Path concept to do the ***ATC*** operation.

C. Data Structure for Dynamic Graphs

Compressed-Sparse Row (CSR) is one of the most widely used representations for sparse data. CSR is applicable in multiple application domains, sparse graph analysis included. One of the most significant limitations of CSR is that it is immutable and cannot be updated. Extending a sparse data format to support dynamic operations, such as edge insertions and deletions, is non-trivial.

In recent years, several data structures and frameworks have been designed to deal with updating the graphs. The original thought was that the graph gets updated by an external source due to some event occurrence. Yet, several recent dynamic graph algorithms show instances where the graph changes due to an internal event. For example, the graph is updated in our ATC algorithm due to new edges getting detected in the anti-section operation. Two other examples include finding a k-truss [47] and k-core decomposition [48] where the graph is updated as part of the algorithm.

The STINGER data structure [18] was first introduced as a dynamic graph structure that can support both temporal and spatial graphs with meta-data (such as vertex and edge types). GraphIn [49] is an incremental only data structure and uses



(a) as-22July20 (b) delaunay-n13 (c) Gnutella09

Fig. 5. Execution time of each phase as a function of the number of new edges found.

both CSR and Coordinate list (COO) formats. Dynamic CSR [50] stores an entire CSR graph for each graph update. This data structure's storage complexity is quite high and requires special traversal primitives for operations that span multiple CSR data structures.

AIMS [51] and FAIMS [52] are two recent dynamic graph data structure focused on improving edge insertion and deletion operations. Both of these data structures use a linked-list of blocks to support dynamic graph operations. The adjacency is split across multiple nodes of a linked-list, making the anti-section (or intersection) operation hard to implement. Furthermore, AIMS has low memory utilization as it allocates all memory on the GPU. Awad *et al.* [53] use a hash table to implement a dynamic graph data structure for the GPU.

cuSTINGER [17] was the first dynamic graph data structure for the GPU. Hornet [16] followed up on cuSTINGER but made significant changes to its data structure to enable tighter memory bounds, better data management, and improved memory reclamation. The Hornet data structure is a dynamic CSR variant.

VII. Conclusion

In this paper, we introduced a novel approach for computing the transitive closure using a dynamic graph. Our ***ATC*** algorithm is simple, scalable, and high-performing. ***ATC*** has several unique features. First, ***ATC*** shows a simple approach for finding new edges using an anti-section operation, similar to an intersection operation, with high efficiency. The similarity between anti-sections and intersections means that our new algorithm can easily benefit from the significant effort in improving graph triangle counting in the last decade. Second, we show that by using a dynamic sparse graph data structure we can reduce the memory footprint, reduce data movement between intermediate matrices, and avoid creating the intermediate matrices. Using Hornet, a scalable and high performing dynamic graph data structure, we showed that edge insertions are efficient and straightforward. Furthermore, the use of bulk edge insertions is another new feature our algorithm shows over past approaches. Third, every computational phase in our algorithm is parallelizable. Because of this, we successfully utilize NVIDIA GPUs and outperform existing CPU and GPU implementations of transitive closures.

Our algorithm's new features are why we can outperform a highly tuned and parallel GraphBLAS implementation (SpareSuite) by $2\times - 3\times$ with 24-cores (48-threads) while using a single NVIDIA Titan V GPU. Our new algorithm was several orders of magnitude faster than a sequential GraphBLAS (SEIGBTL) and the popular NetworkX framework. Lastly, our GPU based implementation of ***ATC*** is faster than the linear algebra-based solution on the same GPU. This highlights that our contribution is exploiting more parallelism through the use of dynamic bulk edge creation and co-designing the algorithm with the use of a dynamic graph data structure.

Acknowledgments

This research was funded in part by NSF grant number CCF-2109988 (Bader).

References

[1] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, *et al.*, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, IEEE, 2016.

[2] A. T. Stephen and O. Toubia, "Explaining the power-law degree distribution in a social commerce network," *Social Networks*, vol. 31, no. 4, pp. 262–270, 2009.

[3] L. A. Adamic, B. A. Huberman, A. Barabási, R. Albert, H. Jeong, and G. Bianconi, "Power-law distribution of the world wide web," *Science*, vol. 287, no. 5461, pp. 2115–2115, 2000.

[4] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of The Internet Topology," in *ACM SIGCOMM Computer Communication Review*, pp. 251–262, 1999.

[5] M. J. Fischer and A. R. Meyer, "Boolean matrix multiplication and transitive closure," in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pp. 129–131, IEEE, 1971.

[6] H. Jagadish, R. Agrawal, and L. Ness, "A study of transitive closure as a recursion mechanism," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 331–344, 1987.

[7] D. L. Monge and T. A. Schultz, "Process for providing transitive closure using fourth generation structure query language (sql)," Oct. 6 1998. US Patent 5,819,257.

[8] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, "A transitive closure algorithm for test generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 7, pp. 1015–1028, 1993.

[9] T. Mak, F. Xia, A. Yakovlev, M. Palesi, *et al.*, "Embedded transitive closure network for runtime deadlock detection in networks-on-chip," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 7, pp. 1205–1215, 2011.

[10] F. Coelho, "Compiling dynamic mappings with array copies," in *Proceedings of the sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 168–179, 1997.

[11] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman, "Transitive closure of infinite graphs and its applications," *International Journal of Parallel Programming*, vol. 24, no. 6, pp. 579–598, 1996.

[12] S. Farheen, N. A. Day, A. Vakili, and A. Abbassi, "Transitive-closure-based model checking (TCMC) in Alloy," *Software and Systems Modeling*, pp. 1–20, 2020.

[13] M. Palkowski and W. Bielecki, "Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing," *BMC bioinformatics*, vol. 18, no. 1, pp. 1–10, 2017.

[14] Y. Ye and J. Talburt, "The effect of transitive closure on the calibration of logistic regression for entity resolution," *Journal of Information Technology Management*, vol. 10, no. 4, pp. 1–11, 2018.

[15] C. Demetrescu and G. F. Italiano, "Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures," *Journal of Discrete Algorithms*, vol. 4, no. 3, pp. 353–383, 2006.

[16] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, (Waltham, MA), 2018.

[17] O. Green and D. Bader, "cuSTINGER: Supporting Dynamic Graph Algorithms for GPUS," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, (Waltham, MA), 2016.

[18] D. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," tech. rep., Georgia Institute of Technology, 2009.

[19] D. P. Martin, "Dynamic shortest path and transitive closure algorithms: A survey," *arXiv preprint arXiv:1709.00553*, 2017.

[20] J. Fox, A. Tripathy, and O. Green, "Improving Scheduling for Irregular Applications with Logarithmic Radix Binning," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, 2019.

[21] T. Davis, Y. Hu, and S. Kolodziej, "The SuiteSparse Matrix Collection," 2018.

[22] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph Structure in the Web," *Computer Networks*, vol. 33, no. 1, pp. 309–320, 2000.

[23] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, "An adaptive parallel algorithm for computing connected components," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2428–2439, 2017.

[24] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection." http://snap.stanford.edu/data.

[25] S. McMillan, "GBTL v3. 0: Primitives for optimized mxm," tech. rep., Carnegie Mellon University, Pittsburgh, PA, 2020.

[26] T. Davis, "SuiteSparse: GraphBLAS," in *High Performance Extreme Computing Conference (HPEC). IEEE*, 2017.

[27] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[28] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM (JACM)*, vol. 9, pp. 11–12, 1 1962.

[29] H. S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Communications of the ACM*, 1975.

[30] V. Pieterse and L. Cleophas, "Benchmarking optimised algorithms for transitive closure," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, pp. 1–10, ACM, 2017.

[31] S.-Y. Kung, S.-C. Lo, *et al.*, "Optimal systolic design for the transitive closure and the shortest path problems," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 603–614, 1987.

[32] A. Velasquez and S. K. Jha, "Brief announcement: Parallel transitive closure within 3d crosspoint memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 95–98, 2018.

[33] E. Nuutila, *Efficient transitive closure computation in large digraphs.* PhD thesis, Helsinki University of Technology, 1998.

[34] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, pp. 146–160, 6 1972.

[35] M. Naumov, A. Vrielink, and M. Garland, "Parallel depth-first search for directed acyclic graphs," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, pp. 1–8, 2017.

[36] F. Bancilhon, "Naive evaluation of recursively defined relations," in *On Knowledge Base Management Systems*, pp. 165–178, Springer, 1986.

[37] B.-F. Wang and G.-H. Chen, "Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 500–507, 1990.

[38] T. Gilray and S. Kumar, "Distributed relational algebra at scale," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 12–22, IEEE, 2019.

[39] S. Samsi, J. Kepner, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, A. Reuther, S. Smith, W. Song, *et al.*, "Graphchallenge. org triangle counting performance," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, IEEE, 2020.

[40] A. Yaşar, S. Rajamanickam, J. Berry, M. Wolf, J. S. Young, and Ü. V. Çatalyürek, "Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments:(update on static graph challenge)," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–4, IEEE, 2019.

[41] O. Green, J. Fox, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. Bader, "Logarithmic Radix Binning and Vectorized Triangle Counting," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, (Waltham, MA), 2018.

[42] J. Fox, O. Green, K. Gabert, X. An, and D. Bader, "Fast and Adaptive List Intersections on the GPU," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, (Waltham, MA), 2018.

[43] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the GPU," in *Proceedings of the ACM Workshop on High Performance Graph Processing*, pp. 1–8, ACM, 2016.

[44] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge Path-Parallel Merging Made Simple," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pp. 1611–1618, 2012.

[45] O. Green, R. McColl, and D. A. Bader, "GPU merge path: a GPU merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 331–340, 2012.

[46] O. Green, P. Yalamanchili, and L. Munguía, "Fast Triangle Counting on the GPU," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pp. 1–8, 2014.

[47] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader, "Quickly Finding a Truss in a Haystack," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, (Waltham, MA), 2017.

[48] A. Tripathy, F. Hohman, D. Chau, and O. Green, "Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure," in *IEEE Proc. Big Data*, (Seattle, WA), 2018.

[49] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, "GraphIn: An Online High Performance Incremental Graph Processing Framework," in *European Conference on Parallel Processing*, pp. 319–333, Springer, 2016.

[50] J. King, T. Gilray, R. M. Kirby, and M. Might, "Dynamic Sparse-Matrix Allocation on GPUs," in *International Conference on High Performance Computing*, pp. 61–80, Springer, 2016.

[51] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, Independent Management of Dynamic Graphs on GPUs," in *International Supercomputing Conference*, pp. 97–119, Springer, 2017.

[52] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 754–766, IEEE, 2018.

[53] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens, "Dynamic graphs on the GPU," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 739–748, IEEE, 2020.