Accelerating Datalog Applications with cuDF

Ahmedur Rahman Shovon*, Landon Richard Dyken*, Oded Green†, Thomas Gilray*, Sidharth Kumar*
* University of Alabama at Birmingham, Birmingham, AL, USA
† NVIDIA

***Abstract*—Datalog, a bottom-up declarative logic programming language, has a wide variety of uses for deduction, modeling, and data analysis across application domains. Datalog can be efficiently implemented using relational algebra primitives such as join, projection and union. While there exist several multi-threaded and multi-core implementations of Datalog targeting CPU-based systems, our work makes an inroad towards developing a Datalog implementation for GPUs. We demonstrate the feasibility of a high-performance relational algebra backend for a subset of Datalog applications that can effectively leverage the parallelism of GPUs using cuDF. cuDF is a library from the Rapids suite that uses the NVIDIA CUDA programming model for GPU parallelism. It provides similar functionalities to Pandas, a popular data analysis engine. In this paper, we analyze and evaluate the performance of cuDF versus Pandas for two graph-mining problems implemented in Datalog, (1) triangle counting and (2) transitive-closure computation.**

I. Introduction

Datalog [1]–[5] is a lightweight logic-programming language for deductive-database systems where queries and database updates are expressed as first-order Horn-clause rules. Running a Datalog program reifies (explicitly deduces) the intentional database (output), which extends data from the extensional database (previously enumerated input data) with all facts transitively derivable via the program's rules. It is a declarative programming language that allows application logic to be specified in a high-level manner, suited for human designers and maintainers, and has thus been applied to a variety of applications including big-data analytics [6]–[8], ML [9] and software analysis [10]–[12].

Datalog programs can be automatically compiled down to low-level relational algebra (RA) primitives for implementation on modern computational systems. Standard RA operations on relations such as selection, join, and union are used in combination to implement efficient kernels that infer new facts from available facts. High-performance operations of RA have the potential to automatically extract data parallelism from applications built on top of declarative languages such as Datalog [1]–[5].

Souffle [13] provides the state of the art implementation for Datalog. It is based on OpenMP and extracts multi-threaded parallelism from CPU-based multi-core systems. Similarly, more recent work [14], [15], [16] aimed toward developing an MPI-based multi-node parallel implementation for Datalog. Both these work target CPU-based systems and do not cater to GPUs. Modern systems (general-purpose and HPC, both) are heading towards heterogeneous computing environments where CPUs are coupled with GPUs. It is essential to develop software systems that can take advantage of the significant parallelism offered by GPUs [17]. This paper makes initial inroads in developing GPU-based implementations for Datalog. We demonstrate the feasibility of implementing Datalog applications using Python's Pandas Dataframe API and further accelerate performance using GPU accelerated cuDF.

Pandas is one of the most popular Python-based data-analysis packages and provides easy-to-use RA functionalities in its DataFrame, a two-dimensional labeled tabular data structure [18], [19]. Pandas has APIs for efficiently joining, aggregating, renaming, deduplicating, and projecting DataFrame rows using CPUs [20]. GPU programming has been accelerating HPC research in recent years by leveraging the massive SIMD parallelism of the GPU. cuDF is a Python package built on Apache Arrow that uses the NVIDIA CUDA programming model for GPU parallelism [21]–[23]. It provides similar functionalities and code syntax to that of Pandas – thus, abstracting developers from dealing with low-level CUDA programming, which often has a steep learning curve. In this paper, we developed Datalog implementations using both Pandas and cuDF libraries, making the following contributions:

1) We demonstrate the feasibility of implementing Datalog applications using the RA primitives of Pandas and cuDF.
2) We evaluate the efficacy of our implementation with two application benchmarks: (a) triangle counting and (b) transitive closure of real graphs. We observe a $147\times$ speedup for triangle counting and $71\times$ speedup for transitive closure computation using cuDF (over pandas).
3) We identify the shortcomings of Pandas and cuDF based Datalog implementations.

II. Datalog

In Datalog, rules can be provided to define relations (tables) in terms of others.

```
B(x,y) :- G(x,y), G(y,x), x<y.
```

Consider the input relation $G(x, y)$ of arity 2 (i.e., with 2 columns), that encodes a graph, where (x, y) corresponds to edges between nodes x and y. The above rule infers a relation B (the *head* clause), which gets a single tuple (x, y) for each bi-directional edge in graph G. The *body* (everything after the symbol ":-") comprises of two subgoals G(x,y) and G(y,x), separated by the symbol ",", which means a logical AND, translating to the join RA primitive. Therefore, in this case, we must effectively join the table with itself; then, a final constraint, $x < y$, filters this output so that edges are added to B only once, in canonical order.

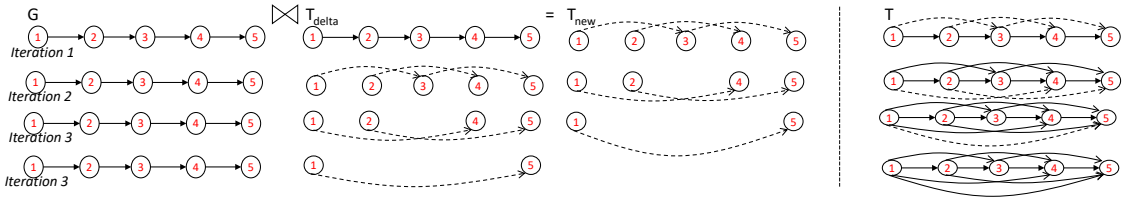

Fig. 1. All iterations of the fixed point loop needed to compute the transitive closure of a chain graph.

In general, evaluation of Datalog rules can be implemented using RA primitives, select, join and projection. A join is performed between two or more subgoals in the body of a rule. A projection is made over the variables in the head of the rule. For recursive rules, fixed-point evaluation is used. The basic idea is to iterate through the rules in order to derive new facts and use these new facts to derive even more new facts until no new facts are derived.

A. Triangle Counting

Triangle counting is one of the most popular graph-mining applications that is used heavily in analyzing social networks. While many tuned algorithms [24] exist to compute triangles, they typically have a steep learning curve. Triangle counting, however, can be easily encoded in Datalog as follows:

```
2cl(x,y) :- G(x,y), x<y.
2cl(x,y) :- G(y,x), x<y.
triangles(x,y,z) :- 2cl(x,y), 2cl(y,z),
                    2cl(x,z).
```

The first two rules infer a relation $2cl$ which prunes away all reflexive vertices from the input graph G and orders all edges. These rules can be compiled down to a copy rule with a filter operator, which translates to the insertion of every element in G into $2cl$, which meets the constraint $x < y$. Furthermore, the third and final rule is compiled down to two join operations. The first join between $2cl(x, y)$ and $2cl(y, z)$ corresponds to joining the graph with itself (joining the graph on the second column with itself on the first column), computing paths of length two, resulting in an intermediate relation, that in turn is joined with $2cl$ to compute all triangles.

B. Transitive Closure

The following recursive Datalog program computes the transitive closure, T, of a graph G:

```
T(x,y) :- G(x,y).
T(x,z) :- T(x,y), G(y,z).
```

The first rule represents a base case that says every x-to-y edge in G implies an immediate x-to-y path in T. The second rule is recursive and must be iterated repeatedly until stabilizing at a value for T that is consistent with all rules. The first rule can be implemented by inserting every element of G into T. The second rule can be implemented by iteration of a kernel function, composed of several relational operations, iterated to a least-fixed-point. One iteration of this function would join T on its second column with G on its first column, yielding all triples (x, y, z) where (x, y) can be drawn from T and (y, z) can be drawn from G. Projection to the set of unique (x, z) tuples, removing the middle column (as a graph, this is removing the intermediate vertex in the discovered path), and unioning this set of tuples with those in T completes one iteration of the second rule.

An example of transitive closure can be seen in Figure 1. The input Graph G is a chain graph with five vertices and four edges. The first iteration computes paths of length 2 – $(1, 3)$, $(2, 4)$ and $(3, 5)$. These new paths are added (unioned) with T, which stores the transitive closure. Due to the recursive nature of the rules and following the semi-naive evaluation method [25], [26] (i.e., incrementalized evaluation), these new paths form the input for the next iteration, which in turn generates paths of lengths 3, $(1, 4)$ and $(1, 5)$. These form the input for the next iteration, and the process continues till a fixed point is reached i.e., no new paths can be found.

III. Datalog-like implementation with Python

Our in-house compiler [14], [15] generates RA kernels in C++ from Datalog. We have used the generated C++ code as a baseline to develop RA kernel for two popular graph mining applications: triangle counting and transitive closure computation in Python using the Pandas and cuDF.

A. cuDF

Pandas is a popular data-analysis package that provides a rich set of pre-built implementations of RA primitives such as join, projection, and union. Pandas, however, is limited in performance as it primarily allows single-threaded execution on the CPU. cuDF is a GPU-based data frame library sharing similar function signatures and API. It is built on the Apache Arrow that provides CPU-GPU interoperability [21]. It allows loading of dataframes onto the GPU for faster I/O, and provides GPU-powered RA functionalities that can be directly applied to dataframes. It reduces the time for copying data, CPU to GPU, and vice versa [22]. Graph analysis is usually applied to large volumes of unstructured data. By reducing the need for data interchange between CPU and GPU devices, cuDF expedites graph mining by significant margins. Alongside the GPU-accelerated I/O operations, cuDF utilizes massive GPU parallelism in RA operations.

B. Triangle counting implementation using relational algebra

Our Datalog implementation of triangle counting needs two copy and two join operations. The first step is to order edges and remove reflexivity, and this is done in two sub-steps. First, copy and filter operations are applied to reverse the columns for rows where the x column is greater than the y column. Then, *drop duplicates* on all rows is called for deduplication. Now, we have a table containing the ordered edges of the

Algorithm 1 Transitive closure computation algorithm

```
1:  procedure TRANSITIVECLOSURE(Graph G)
2:      result ← G
3:      R ← Rename(G)
4:      do
5:          newTriplets ← Join(R, G)
6:          inferredPaths ← Deduplication(Projection(newTriplets))
7:          oldLength ← Length(result)
8:          result ← Deduplication(Union(result, inferredPaths))
9:          currentLength ← Length(result)
10:         R ← Rename(inferredPaths)
11:     while oldLength ≠ currentLength
12:     return result
13: end procedure
```

graph, and encoded as xy. We then copy this table to two others using the *rename* function to obtain xz and yz. From there, *merge* is called between xy and yz to join on the y column, creating a table xyz which contains entry rows (x, y, z) where edges exist in our input graph between x and y, and y and z. Lastly, we use *merge* between xyz and xz to join on the x and z columns, giving us the output rows (x, y, z) where edges also exist between x and z. Because we have checked the connection between all three vertices x, y, and z, we know the rows of this output are the triangles of our input graph.

C. Transitive closure implementation using relational algebra

We implement an algorithm for computing transitive closure using RA APIs provided by cuDF and Pandas. Our transitive closure algorithm is shown in Algorithm 1. We read the dataset using the *read csv*. We use the *merge* on the common column for the join operation of two relations. For the projection operation, we invoke *drop* chaining with *drop duplicates*. Similarly, we use the *concat* to apply union operation.

As described in Section II.B, transitive closure computation is recursive in nature, which translates to computing the RA kernel in a fixed point loop. We can see that the fixed point loop is represented in the form of the *do while* loop at line 4. A drawback of using these pre-baked implementations is that these RA operations, like join and projection in lines 5 and 6, cannot be fused together. Ideally, they can be combined in one GPU routine. However, the current APIs do not support that operation. Another important aspect is that of deduplication, as paths can be found in multiple ways and therefore, there will be multiple copies of the same path. It is essential to deduplicate within every iteration to ensure that we do not perform unnecessary computation.

IV. Evaluation

We perform three sets of experiments to evaluate the efficacy of our cuDF implementation. First, we compare the standalone execution time of RA operators such as rename, inner join, and union and other non-RA operations like deduplication and file I/O for cuDF and Pandas. Following this, we evaluate the performance of our applications: triangle counting and transitive closure on real datasets using Pandas and cuDF.

A. Experimental setup

1) System and configuration: We ran our experiments on the ThetaGPU supercomputer of Argonne Leadership Computing Facility [27]. ThetaGPU is comprised of 24 NVIDIA DGX A100 nodes. Each DGX A100 node comprises eight NVIDIA A100 Tensor Core GPUs. 22 of the 24 nodes have 320 GB of GPU memory each and remaining two nodes have 640 GB of GPU memory. Each node has dual AMD EPYC 7742 processors running at 3.31GHz with 64 cores per processor for a total of 128 cores. The experiments are run using Ubuntu 20.04LTS operating system. We use cuDF and Pandas Python packages inside conda environment with CUDA as GPU programming model. cuDF runs on a single GPU by default, and thus our experiments were restricted to using a single GPU (out of the 8 present on every node).

TABLE I
DATASETS USED IN OUR EXPERIMENTS

Graph	Type	$\|V\|$	$\|E\|$
p2p-Gnutella09	Directed	8,114	26,013
p2p-Gnutella04	Directed	10,876	39,994
Skitter	Undirected	1,696,415	11,095,298
roadNet-CA	Undirected	1,965,206	2,766,607
roadNet-TX	Undirected	1,379,917	1,921,660
roadNet-PA	Undirected	1,088,092	1,541,898
SF.cedge	Undirected	1,74,955	2,23,001
cal.cedge	Undirected	21,048	21,693
TG.cedge	Undirected	18,263	23,874
OL.cedge	Undirected	6,105	7,035

2) Datasets: Our evaluation includes real-world graph datasets from the Stanford large network dataset collection and road network real datasets collection [28], [29]. Our datasets include both directed and undirected graphs. Table I shows the properties of the datasets. $|V|$ indicates number of nodes and $|E|$ indicates number of edges in the graph.

B. Standalone relational operations

We compare the execution time for key operations such as file I/O, rename, inner join, deduplication, and union operations for cuDF and Pandas. Figure 2 shows the time comparisons for *roadNet-CA* dataset mentioned in Table I. We see 8.9x speedup for file I/O, 34.1x speedup for join, 52.2x speedup for rename, 52.4x speedup for deduplication, and 7.1x speed up for union operation in cuDF over Pandas. These results demonstrate the importance of cuDF, and assures speedups for Datalog like applications built on top of these operators.

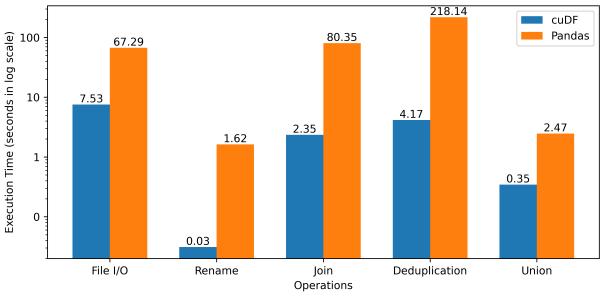

Fig. 2. Time comparison between cuDF and Pandas for relational operations

C. Triangle counting

Figure 3 shows the time comparisons for selected datasets for triangle counting, while Table II shows the speedup and output triangle counts. As can be seen, cuDF greatly outperforms Pandas for datasets with a large number of rows or

TABLE II
SPEEDUP OF CUDF OVER PANDAS FOR TRIANGLE COUNTING

Dataset	Triangles	cuDF(s)	Pandas(s)	Speedup
Skitter	28,769,868	1.0340	152.7032	147.7x
roadNet-CA	120,676	0.1593	3.0106	18.9x
roadNet-TX	82,869	0.1180	1.9466	16.5x
roadNet-PA	67,150	0.1084	1.4777	13.6x
SF.edge	4,036	0.0877	0.1482	1.7x
p2p-Gnutella09	2,354	0.0486	0.0256	0.5x

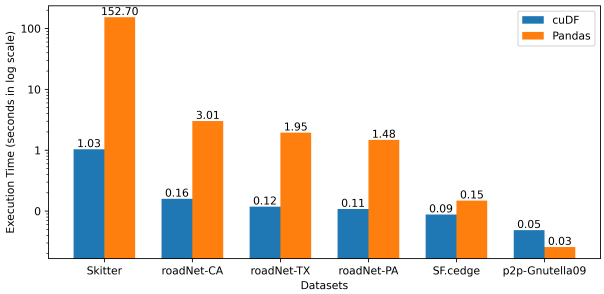

Fig. 3. Time comparison between cuDF and Pandas for triangle counting

TABLE III
SPEEDUP OF CUDF OVER PANDAS FOR TRANSITIVE CLOSURE

Dataset	TC size	Iterations	cuDF(s)	Pandas(s)	Speedup
SF.cedge	80,498,014	287	64.235	4582.067	71.3x
p2p-Gnutella09	21,402,960	20	3.881	167.143	43.1x
p2p-Gnutella04	47,059,527	26	14.104	569.249	40.4x
cal.cedge	501,755	195	3.883	5.769	1.5x
TG.cedge	481,121	58	1.191	1.400	1.2x
OL.cedge	146,120	64	0.557	0.474	0.9x

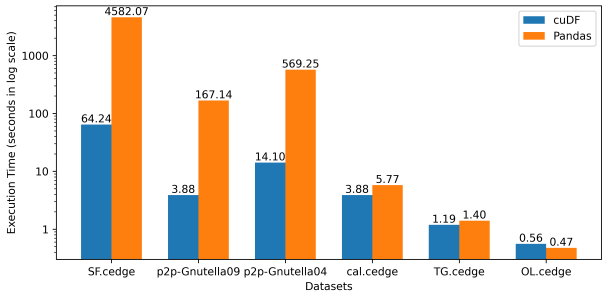

Fig. 4. Time comparison between cuDF and Pandas for transitive closure

output triangles, with a maximum speedup of 147.7x for the Skitter dataset containing 28,769,868 triangles. This is because the triangle counting algorithm consists of deduplication, two renames, and two join operations, all of which are shown in Figure 2 to be much faster in cuDF than Pandas when the size of the graph is as large as *roadNet-CA*. The trend of speedup in Table II shows that the benefit of the GPU-utilizing backend of cuDF offers a greater speedup over Pandas directly correlated to the size of the input graphs for triangle counting.

D. Transitive closure comparison

Figure 4 shows the time comparisons for selected datasets for transitive closure, while Table III shows the speedup and output graph size. The TC size indicates the size of the transitive closure operation for the graph. The number of iterations to compute the TC is shown in the iterations column. The speedup column presents the speedup of execution time for cuDF over Pandas. Higher speedups of the first three graphs (71x, 43x and 40x) can be attributed to their relatively larger workload, as indicated by their final TC size (indicative of workload). A large workload ensures proper saturation and utilization of the GPU, translating to higher speedups.

E. Limitations

The APIs such as read, merge, drop, and concat in cuDF and Pandas are predefined. These APIs require a specific set of parameters and return results in a specific format. RA kernels of Datalog would benefit from fusing of RA operations, such as joins, projection and deduplication, however, cuDF API does not allow such operations. These APIs must be invoked sequentially while storing the intermediate result – adding memory and computation overhead. For example, in a hand tuned implementation, the two consecutive joins in triangle counting could potentially be performed in one single operation (nested loops) without having to explicitly materialize the results of the first join operation. We also run into memory overflow errors while performing transitive closure computation for graphs with several million edges in cuDF implementation. These limitations suggests using a scalable backend for Datalog applications that is optimized for a multi-node, multi-GPU environment.

V. CONCLUSION AND FUTURE WORK

In this paper, we demonstrate the feasibility of implementing Datalog applications using GPU-based cuDF relational algebra primitives and show performance gain (up to $147\times$ for triangle computation and $71\times$ for transitive closure computation). We test the performance gain using both directed and undirected graphs. We identify several bottlenecks for these implementations. Given the good performance improvement we got with cuDF in a relative short amount of time we are now eager investigating several other facets: 1) how good does our Python implementation of Datalog applications compare to native counterparts (comparing our Pandas solution with NetworkX [30] and comparing our cuDF solution with cuGraph [21] and Hornet [31] and 2) extend our solution to using Dask/cuDF for multi-GPU applications to allow for improved scaling. As part of our continued investigation, we will attempt to identify missing algorithmic components in Pandas and cuDF necessary for a broader set of applications. Our code, data, and documentation is open-source and available at https://github.com/harp-lab/GPUJoin.

VI. ACKNOWLEDGEMENT

This work was funded in part by NSF RII Track-4 award 2132013 and NSF collaborative research award 2217036. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the ThetaGPU supercomputer located at the Argonne National Laboratory.

REFERENCES

[1] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1371–1382. [Online]. Available: https://doi.org/10.1145/2723372.2742796

[2] S. Ceri, G. Gottlob, L. Tanca *et al.*, "What you always wanted to know about datalog(and never dared to ask)," *IEEE transactions on knowledge and data engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[3] O. De Moor, G. Gottlob, T. Furche, and A. Sellers, *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Springer, 2012, vol. 6702.

[4] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: an interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 1213–1216.

[5] J. D. Ullman, *Principles of database systems*. Galgotia publications, 1983.

[6] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker *et al.*, "Demonstration of the myria big data management service," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 881–884.

[7] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 278–289.

[8] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1135–1149.

[9] R. J. Mooney, "Inductive logic programming for natural language processing," in *International conference on inductive logic programming*. Springer, 1996, pp. 1–22.

[10] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009, pp. 243–262.

[11] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis, "Set-based pre-processing for points-to analysis," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 253–270.

[12] E. Hajiyev, M. Verbaere, and O. De Moor, "Codequest: Scalable source code queries with datalog," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 2–27.

[13] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.

[14] S. Kumar and T. Gilray, "Distributed relational algebra at scale," in *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, 2019.

[15] T. Gilray, S. Kumar, and K. Micinski, "Compiling data-parallel datalog," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, 2021, pp. 23–35.

[16] S. Kumar and T. Gilray, "Load-balancing parallel relational algebra," in *International Conference on High Performance Computing*. Springer, 2020, pp. 288–308.

[17] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[18] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[19] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[20] D. Y. Chen, *Pandas for everyone: Python data analysis*. Addison-Wesley Professional, 2017.

[21] R. D. Team, *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. [Online]. Available: https://rapids.ai

[22] A. Fender, B. Rees, and J. Eaton, "Rapids cugraph," in *Massive Graph Analytics*. Chapman and Hall/CRC, 2022, pp. 483–493.

[23] O. Green, Z. Du, S. Patel, Z. Xie, H. Liu, and D. A. Bader, "Anti-section transitive closure," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2021, pp. 192–201.

[24] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, p. e1226, 2018.

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

[26] M. Shaw, P. Koutris, B. Howe, and D. Suciu, "Optimizing large-scale semi-naïve datalog evaluation in hadoop," in *International Datalog 2.0 Workshop*. Springer, 2012, pp. 165–176.

[27] A. Leadership Computing Facility, "Argonne leadership computing facility," 2022. [Online]. Available: https://www.alcf.anl.gov/support-center/theta-gpu-nodes

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[29] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *International symposium on spatial and temporal databases*. Springer, 2005, pp. 273–290.

[30] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[31] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.