Towards Iterative Relational Algebra on the GPU

Ahmedur Rahman Shovon[1], Thomas Gilray[1], Kristopher Micinski[2], and Sidharth Kumar[1]

[1]University of Alabama at Birmingham
[2]Syracuse University

Abstract

Iterative relational algebra (RA kernels in a fixed-point loop) enables bottom-up logic programming languages such as Datalog. Such declarative languages are attractive targets for high-performance implementations of relational data analytics in fields such as graph mining, program analysis, and social-media analytics. Language-level constructs are implemented via high-performance relational algebra primitives (e.g., projections, reorderings, and joins). Such primitives would appear a natural target for GPUs, obtaining high throughput on large datasets. However, state-of-the-art Datalog engines are still CPU-based, scaling best between 8–16 threads. While much has explored *standalone* RA operations on the GPU, relatively less work focuses on *iterative* RA, which exposes new challenges (e.g., deduplication and memory management). In this short paper, we present a GPU-based hash-join implementation, leveraging (a) a novel open-addressing-based hash table implementation, (b) operator fusing to optimize memory access and (c) two variant implementations of deduplication. To evaluate our work, we implement transitive closure using our hash-join-based CUDA library and compared its performance against cuDF (GPU-based) and Soufflé (CPU-based). We show favorable results against both, with gains up to 10.8× against cuDF and 3.9× against Soufflé.

1 Introduction

High-performance iterative relational algebra (RA) has the potential to automatically extract massive data-parallelism from applications built on top of bottom-up logic programming languages such as Datalog [2, 6, 9, 21, 38]. Datalog is a *declarative* logic programming language that has been applied to a wide variety of applications such as big-data analytics [17], graph mining [26, 33], and program analysis [5, 34]. The goal of declarative languages is for a user to provide a few understandable and compact rules that define a solution while the language automatically extracts an operational approach for computing the solution. Standard bottom-up Datalog solvers do just this: rules are implemented via standard RA operations and combined into kernels that infer new facts from discovered facts in a fixed-point loop.

Modern Datalog applications scale to extreme input-relation sizes (billions of rows/facts, tens of gigabytes of data), and thus highly parallel implementations are increasingly valuable. Unfortunately, however, modern CPU-based Datalog implementations have hit scalability walls due to their use of locking shared-memory data-structures—in our experiments, scalability for Soufflé (a best-in-class solver) peaks at roughly 16 threads. By contrast, modern GPUs offer (tens of) thousands of data-threads of computation in their high-throughput SIMD architecture. While there has been a plethora of work discussing implementation of standalone joins [20, 24, 30–32, 36] on the GPU, not much work tackles the problem of iterative joins, especially in the context of the modern GPU architectures. Iterative joins, fused with other relational operations such as union and projection, needed in the context of Datalog engines, add extra layers of complexity such as having to deal with low-level memory management, performing deduplication and mantainance for index data structures.

In this paper, we present key innovations which have allowed us to build scalable GPU-based implementations of Datalog-style declarative programs. Specifically, we implement the classic Datalog problem of finding all reachable paths (i.e., transitive closure) of a graph—a simple case of feature extraction. This problem entails implementing joins along with other RA operations like union and projection iteratively in a fixed-point loop. With this work, we make an important first step towards developing a complete GPU based datalog engine. Specifically, we make the following novel contributions to literature:

1. Developed a high-performance GPU-based hash table tailored to relational data; the hash table is used to accomplish binary hash joins between relations.
2. Implemented RA-operation fusion (e.g. join and projection) to improve memory and computation footprint.
3. Implemented deduplication using two techniques (a) sort

and unique (using thrust [4]), and (b) merge (two sorted lists) and unique. Both, key in facilitating iterative RA.
4. Evaluated our performance by comparing it against state of art openMP-based implementation, Soufflé and a cuDF implementation. We outperform both for almost all graphs and achieve speedups up to 3.9× over Soufflé and up to 10.8× over cuDF.

2 Related Work and Background

RA on GPUs Previous efforts in parallelizing RA have mostly focused on stand-alone implementations of select few RA primitives, such as join. Join is the most complex primitive to implement as its output size is not known in advance and depends on the characteristics of the input data and relations must be sorted, or stored in an index, for efficient joins to be possible. The most common algorithms for implementing joins are hash-joins and merge-sort join [3]. These algorithms have been extensively studied for shared memory systems, being parallelized for both GPUs [16, 19, 20, 24, 30–32, 36] and multi-core CPUs using openMP [23]. MPI-based distributed join operations have also shown promise in several recent studies [10, 13, 14, 25, 26]. However, unlike distributed join implementations, most extant GPU-based implementations do not maintain an order in their relations [39]. This poses a problem in some iterated relational algebra algorithms (such as transitive closure computation) as they require the join results to be sorted. If join results are sorted by default, we can avoid costly operations like sorting the entire result at each iteration of the algorithm, which impacts the overall performance of all iterations and compounds across the fixed-point loop. Additionally, off-the-shelf Python libraries also provide RA primitives to perform iterated join operations [35], but using the predefined methods does not allow fusing RA operations and creates memory overhead storing all intermediate results during iterated RA operations.

Datalog and iterated RA In Datalog, rules can be provided to define relations (tables) in terms of others. The following Datalog program inductively defines the transitive closure, T, of an input graph G using two rules: $T(x,y)$:– $G(x,y)$. and $T(x,z)$:– $T(x,y), G(y,z)$. 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 says: every x-to-y path in T that can be extended with a y-to-z edge in G, implies an extended x-to-z path. 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, a simple copy or union operation. The second rule can be implemented by iteration of a kernel function, composed of several RA 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. The output (newly discovered facts) acts as the input for the following iteration, and the process continues till all reachable paths are discovered (see Figure 1).

3 Standalone Join

We first implement a standalone join operation between two input relations. A low-level programming model such as CUDA allows us to control the memory hierarchy and fuse operations together. We create a static hash table on the input relation and then use a hash-join-based approach to join the relations.

Representation We have developed a novel GPU-based hash table from scratch to meet requirements pertinent to our system. We extend the hash-join-based approach to support the relational data type specific to our application (2-ary column). This is done by implementing an efficient hash table that effectively supports search and can be easily linearized to a compact 1D array (required for deduplication)– both essential in implementing iterative joins.

To facilitate seamless hash-joins between relations, we use the entries of the join column of every relation as the key of the hash table. In particular, the join column of a relation is used as the *key*, which *maps* to a set of values (corresponding to the non-join column values). For example, for a relation that is hashed on the first column, with tuples $(1,2)$, $(1,3)$, and $(1,4)$ the key corresponds to 1 and the set of values would correspond to $2,3,4$. Note that, our hash-table uses only the join column as the key for hashing– making it easy to facilitate fast look-ups, needed for joins. Furthermore, to simplify our development (targeting graph analytic applications), we assume all base values are 32 bits wide.

All hash tables must support collision resolution. Two popular schemes include separate chaining and open addressing. However, GPUs face intrinsic difficulties in dealing with linked data, and we instead use an open-addressing-based hash table consisting of a fixed-size buffer. In our design, collision resolution is accomplished via linear probing, which (in our experiments) shows better cache performance than quadratic and double-hashing techniques. Our implementation uses the `Murmur3` hash, a popular implementation that performs well on open-source benchmarks [22]. The hash is calculated using a combination of bit shifts, multiplications, and XOR operations. We obtain a hash-table build rate of 4 billion keys per second for a string graph and 400 million keys per sec for a random graph (see Sec. 5 for more details).

Algorithm 1 Hashjoin based transitive closure computation algorithm. Blue boxes indicate join, orange indicates union, green indicates deduplication, and red indicates clearing.

```
1:  procedure TRANSITIVECLOSURE(Graph G)
2:      R ← HashTable(G)
3:      result ← Sort(G)
4:      T_Δ ← G
5:      repeat
6:          joinSizePerRow ← JoinSize(R, T_Δ)
7:          joinOffset ← Scan(joinSizePerRow)
8:          Initialize(joinResult, totalJoinSize)
9:          joinResult ← Join((R, T_Δ), joinOffset)
10:         joinResult ← Sort(joinResult)
11:         joinResult ← RemoveDuplicates(joinResult)
12:         totalUniqueJoinSize ← Size(joinResult)
13:         FreeMemory(T_Δ)
14:         T_Δ ← Copy(joinResult, totalUniqueJoinSize)
15:         unionSize ← resultSize + totalUniqueJoinSize
16:         Initialize(unionResult, unionSize)
17:         unionResult ← MergeSortedArrays(result, joinResult)
18:         unionResult ← RemoveDuplicates(unionResult)
19:         uniqueUnionSize ← Size(unionResult)
20:         oldUnionSize ← Size(result)
21:         FreeMemory(result)
22:         result ← Copy(unionResult, uniqueUnionSize)
23:         FreeMemory(joinOffset)
24:         FreeMemory(joinResult)
25:         FreeMemory(unionResult)
26:     until oldUnionSize ≠ uniqueUnionSize
27:     FreeMemory(R)
28:     FreeMemory(result)
29:     FreeMemory(T_Δ)
30:     return result
31: end procedure
```

In an open-addressing-based hash table, the load factor measures a table's sparsity, Load factor$(\alpha) = size(input)/size(hashTableSize) \leq 1$. Intuitively, a lower load will yield fewer collisions (and thus higher performance) [11]. In experiments, we use load factors 0.1 and 0.4 to pre-allocate our hash table as the nearest (larger) power of two (called *hashTableSize*); this permits us to replace the more expensive modulo operation in our hash function with an efficient binary AND operation. We create the hash table as a strided array of structures (*Entity*) with 32-bit integers *Key* and *Value* as fields. The hash table is initialized with a sentinel value (-1) to indicate an empty value. While doing bulk insertions in the hash table, CUDA threads search for the index of each tuple they are assigned, using the hash function. If the position is available (i.e., -1), we insert the tuple in the hash table using CUDA's atomic compare-and-swap operation (*atomicCAS*). If the position is already occupied, we search for the next available position using linear probing. In CUDA, if each thread only operates on one element of the input array, it can lead to problems if the input is larger than the number of available threads. To address this, we use a *grid-stride loop*, which allows a single thread to traverse multiple elements of the input array by incrementing its index by a stride value that is determined by the grid size and block size. Each CUDA thread performs build hash table computation on one grid size ($blockDim.x * gridDim.x$) at a time to provide maximum memory coalescing.

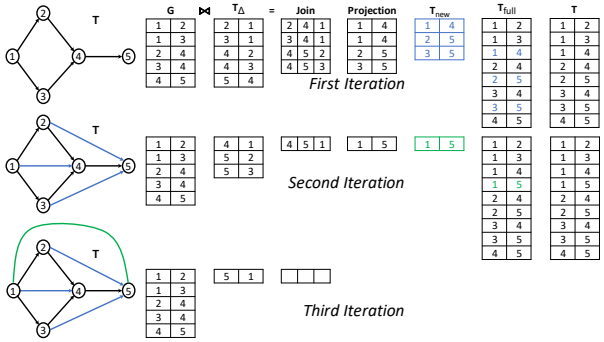

Figure 1: An example of the iterated joins on transitive closure computation of a graph

Hash-Join Phases We perform a join operation to get the join result of the hash table and input relation of *Entity* structure element with two 32-bit integers (*Key* and *Value* as structure members). We use two passes to compute the join result. These two passes are carried out by two CUDA kernels. First, we initialize an array (*offset*) with the size equal to the input relation. In the first phase, each CUDA threads searches for a total of n keys of the input relation in the hash table where n is the size of one grid($blockDim.x * gridDim.x$). If it finds a matching key in the hashtable, it increases the counter of the *offset* array for that input element. Thus, we calculate the number of matches for each key of the input relation using this CUDA kernel. We use thrust library's *reduce* and *exclusive_scan* APIs with device execution policy to get the total amount of matches and the exclusive prefix sum of the *offset* array. As the *offset* array reveals the size of the matches for each key of the input relation, the second pass inserts the joined columns to the join result array using another CUDA kernel. In the second pass, we do not insert the join column to fuse the join and projection operation in one kernel invocation. Both of these kernels use offset-based calculation without using any atomic operations or barrier synchronization, which are computationally costly. On top of that, our kernels use grid-stride loops to distribute the workload equally to the available CUDA threads. It also eliminates the branch divergence problem, where some CUDA threads could have more workloads than others.

4 Iterated Joins on the GPU

Implementing iterated joins requires the allocation of extra buffers to materialize intermediate results. Additionally, these intermediate results must be deduplicated between each RA iteration to ensure tractability. In this section, we demonstrate our approach to efficient iterated joins, using transitive closure as an illustrative example; our techniques generalize to other problems using finite-domain Datalog.

Transitive closure: Transitive closure is operationalized as iterated joins between a monotonically-growing set of transitive edges, T, and an extensional database of edges, G. With each iteration, new paths of increasing lengths are discovered (monotonically extending T) until all reachable paths are found and execution terminates. In practice, efficient implementations of Datalog employ semi-naïve evaluation [1], tracking a *frontier* of new facts rather than all-yet-seen facts. This is implemented by distinguishing multiple run-time tables for each syntactic relation: T_Δ, T_{full}, and T_{new}. T_Δ tracks the most-recently-discovered (last iteration's) transitive edges in T—T_Δ is merged into T_{full} after each iteration. Facts discovered during the *current* iteration are accumulated into T_{new}, which is swapped into T_Δ between iterations.

Figure 1 visualizes the execution of transitive closure on a line graph consisting of five nodes. As a preparatory step before the first iteration, T_Δ is populated by G. The first iteration joins G and T_Δ, yielding four (intermediate) triples. This is followed by projection of the join column, which results in a duplicate edge $(1,4)$. We deduplicate by sorting the results and applying consecutive deduplication (Thrust's unique, which required a preceding stable sort) to produce unique inferred paths in T_{new}. As a data-structure invariant, tuples are sorted (using the natural lexicographic sort): this enables merging T_{new} into T using a *single* scan of T_{new}. Next, we union the merged relation and generate the union result in T_{full}. This graph is visualized in the middle left side of Figure 1, where blue edges indicate the new unique paths found from the first iteration. We update the graph relation T_Δ with T_{new} for the next iteration. We continue this iterated join operations until we cannot find any new path or the size of T does not change after the deduplication of the merged relation.

Algorithm 1 formalizes our implementation of transitive closure in CUDA. This algorithm uses a combination of library functions illustrating the cruxes of Datalog on the GPU: joins (boxed in blue), union (orange), deduplication (green), and clearing (red). We begin by establishing the initial invariants: building an initial *result* by sorting G, and copying G into T_Δ. Next, we iterate until we reach a fixed-point; at the end of each iteration we compare the number facts in the next T with the number in the current T—when no new facts are added, the algorithm terminates at a fixed-point.

Join operation in transitive closure The join operation uses the single hash-join operation in a fixed-point iteration described in Section 3. We create the hash table (R in Algorithm 1) only once and repeatedly use it in the iterated hash join. The relation T_Δ is initialized with input relation G. The two-pass approach (Sec. 3) calculates the join result size (*joinOffset*) for each record of T_Δ and then inserts the join result in the *joinResult* array. Our algorithm fuses the join and projection operations together and thus reduces time and memory consumption for the iterated join (by up to 5%). This optimization is novel from an iterated join on the GPU perspective.

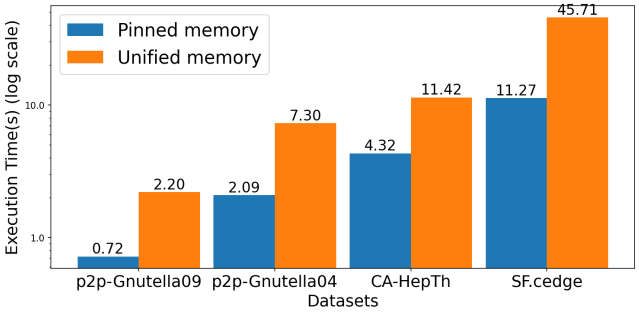

Figure 2: Time comparison between CUDA's pinned memory and unified memory model for hash join based TC computation. (log-scale for Y-axis)

Deduplication of the join result and merged result We use the key insight that some relational data can be permanently maintained in sorted order while some cannot and use this information to implement two kinds of deduplication techniques. Duplicate tuples are generated in two segments; after the join and projection operation and after the union operation. We use thrust library's *sort* and *unique* API to remove duplicates from the join result. The *sort* API first sorts the join results, and then the *unique* API removes the consecutive duplicate elements from the array. To deduplicate the union result, we initialize the *result* array with the input graph G and sort it. To remove duplicates from the union result ($result \cup T_\Delta$), we merge the sorted arrays ($result$, T_Δ). Then we apply the thrust's *unique* API to remove the consecutive duplicates from the merged result. This merging step eliminates the requirement of sorting the large *result* array before applying the deduplication process.

Memory management The CUDA programming model provides four memory allocation schemes: pageable memory, pinned memory, mapped memory, and unified memory [7]. Pageable memory involves two transfers: from host pageable memory to temporary pinned memory on the host, and then to device memory. Pinned memory initializes data on the host's pinned memory, requiring only one transfer to send it to device memory [18]. Mapped memory maps pinned memory in the device address space, avoiding host-to-device memory copying but increasing program processing time. Unified memory creates a managed memory pool for accessing data from both host and device using the same address, but introduces supplementary operations for memory management. For both the single join operations (Sec. 3) and iterated join operations (Sec. 4), we utilize the CUDA pinned memory model. We keep only two arrays (input relation and the final result) in the host memory, and all intermediate buffers are kept in the device memory without any need for data transfer

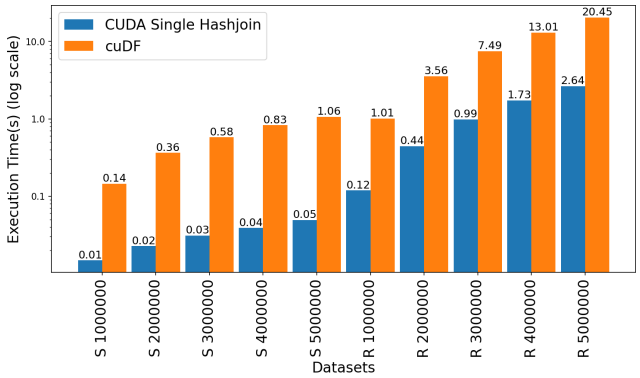

Figure 3: Time comparison between single join operation using CUDA and cuDF. R indicates Random, and S indicates String graph. (log-scale for Y-axis)

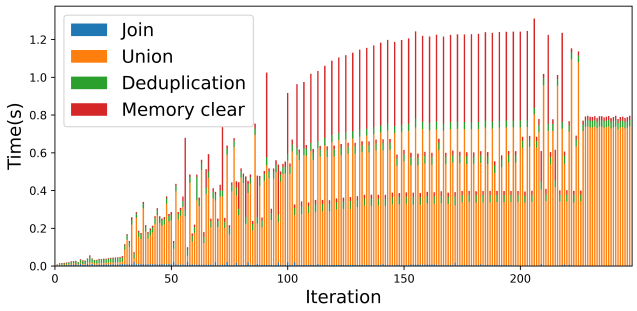

Figure 4: Our time breakdown for *fe_ocean* graph (CUDA outperforms Soufflé by 3.9×)

between the host and device memory for intermediate results. We ensure the implementations have no memory leakage using *cuda-memcheck* and *compute-sanitizer* API provided by CUDA library.

5 Evaluation

We perform a series of experiments to evaluate the performance of our iterative hash-based join implementation. We begin by evaluating the performance tradeoffs offered by using different memory management schemes of CUDA. Following which, we study the performance of a hash-join in isolation, comparing it against state-of-art join implementation of the cuDF library – part of NVIDIA's rapids framework that uses NVIDIA CUDA programming model for GPU parallelism [12, 15, 37]. Finally, we evaluate the performance of our iterative joins, by computing the transitive closure of a range of graphs. We compare our performance against Souffle, a state of art openMP-based library for performing iterated joins, and our cuDF-based implementation.

Experiment platform and datasets We conduct our experiments on the ThetaGPU supercomputer of Argonne National lab [27]. It has 24 NVIDIA DGX A100 nodes with eight NVIDIA A100 Tensor Core GPUs per node. Each node features two AMD EPYC 7742 processors with 3.31GHz clock speed and a total of 128 cores.

cuDF package was installed on a Python conda environment, and we developed our code using CUDA version 11.4. We use Souffle version 2.3 with 128 threads in ThetaGPU. As cuDF and our CUDA experiments use only one GPU device (single-gpu benchmark) we use a single GPU node from ThetaGPU. The single GPU node has 108 multiprocessors on device (SM) and we use $3,456$ (32×108) blocks per grid and 512 threads per block for each of the CUDA kernels.

To evaluate the experiments, we use real-world and synthetic graph datasets from the Stanford large network dataset collection, SuiteSparse matrix collection, and road network real datasets collection [8,28,29]. Table 1 shows all our graphs used along with performance results.

Memory optimization We evaluated the pinned memory, and unified memory as memory allocation schemes for transitive closure computation for several real-world graphs mentioned in Table 1 [7]. Figure 2 shows the time comparison between these two memory allocation schemes for directed and undirected graphs. We observe 2.6× to 4.1× speedup for transitive closure computation using the pinned memory over the unified memory model. However, we see that the unified memory model can handle graphs with larger TC size (e.g. dataset $p2p-Gnutella31$ from Table 1) without getting a run time error (out of memory error) as unified memory can oversubscribe the GPU global memory since CUDA version 8. Therefore, the graph $p2p-Gnutella31$, which has a higher workload per iteration overflows the GPU global memory (with pinned memory). For a single GPU-based implementation, this is an expected result.

Single hash-join performance To benchmark the single hash-join implementation (described in Sec. 3), we use two types of synthetic datasets; random graphs (numbers in the range of 0 to 32,767) and string graphs (e.g. $(1 \rightarrow 2), (2 \rightarrow 3), (3 \rightarrow 4)$). Both types of graphs have edges between 1 million to 5 million. Figure 3 shows the time performance comparison between cuDF and our standalone join.

For a random synthetic graph, we can build a hash table at a rate of 400 million keys per second. For the string graph, the build rate goes up to 4 billion keys per second. These two graph types invoke the two ends of the performance spectrum, with the random graph leading to tons of inefficient matches (worst case) and the string graph leading to perfectly aligned matches (best case). We observe up to 21× speedup for single hash-join computation for the string graph using our CUDA-based implementation over the cuDF join implementation. We notice that the increase in speedup is directly related to the size of the input relation for string graphs. For the random graph, we see 7.6× to 8.4× speedup using the CUDA-based

Table 1: Transitive closure performance using Hashjoin based CUDA, Souffle, and cuDF implementation. CUDA implementation uses 3,456 blocks and 512 threads per block. The Soufflé implementation uses 128 threads. Type *U* and *D* indicate undirected and directed graphs.

Dataset	Type	Rows	TC size	Iterations	CUDA Hashjoin(s)	Soufflé(s)	cuDF(s)
fe_ocean	U	409,593	1,669,750,513	247	138.237	536.233	Out of Memory
p2p-Gnutella31	D	147,892	884,179,859	31	Out of Memory	128.917	Out of memory
usroads	U	165,435	871,365,688	606	364.554	222.761	Out of Memory
fe_body	U	163,734	156,120,489	188	47.758	29.070	Out of Memory
loc-Brightkite	U	214,078	138,269,412	24	15.880	29.184	Out of Memory
SF.cedge	U	223,001	80,498,014	287	11.274	17.073	64.417
fe_sphere	U	49,152	78,557,912	188	13.159	20.008	80.077
CA-HepTh	D	51,971	74,619,885	18	4.318	15.206	26.115
p2p-Gnutella04	D	39,994	47,059,527	26	2.092	7.537	14.005
p2p-Gnutella09	D	26,013	21,402,960	20	0.720	3.094	3.906
wiki-Vote	D	103,689	11,947,132	10	1.137	3.172	6.841
cti	U	48,232	6,859,653	53	0.295	1.496	3.181
delaunay_n16	U	196,575	6,137,959	101	1.137	1.612	5.596
luxembourg_osm	U	119,666	5,022,084	426	1.322	2.548	8.194
ego-Facebook	U	88,234	2,508,102	17	0.544	0.606	3.719
cal.cedge	U	21,693	501,755	195	0.489	0.455	2.756
TG.cedge	U	23,874	481,121	58	0.198	0.219	0.857
wing	U	121,544	329,438	11	0.085	0.193	0.905
OL.cedge	U	7,035	146,120	64	0.148	0.181	0.523

implementation over the cuDF based implementation.

Iterated hash-join We compare our iterated hash-join-based transitive closure computation with state-of-the-art Souffle and Nvidia's cuDF library in Table 1. The benchmark results are based on multiple runs (at least 10), with the average being reported. The variance between runs was negligible (maximum standard deviation is <1%). For 15 out of 19 graphs, our CUDA-based implementation outperforms Souffle's implementation (128 threads). For the graph with the largest TC (*fe_ocean*, 1.6 billion edges), we observe a speedup of 3.9× over Souffle. We observe a speedup of 10.8× over cuDF implementation. Moreover, the cuDF implementation gets out of memory error several times where the CUDA based implementation is able to compute the transitive closure of those graphs using the same experimental setup. Additionally, we fused the projection operation with join operation in CUDA implementation, which is not possible in cuDF-based implementation. Thus, it shows both time and space performance enhancement of our iterated hash-join-based transitive closure computation. To better understand the time consumption for each of the individual operations (mainly join, union, deduplication, and memory clear) at the iteration level, we break down the operations at the granular level. One such granular benchmark is shown in Figure 4. We notice that the deduplication and union operation takes more time than the other operations. Also, as the number of join results is smaller in the first few iterations, it takes significantly less time in those iterations. For the graph (*usroads*), where we under-perform compared to souffle, our hypothesis is that we are not saturating the GPU as this graph has an increasing number of iterations (606) and less work per iteration.

6 Conclusion

We explored the issues for iterated operations such as inefficient operation fusion, GPU memory management, and facts deduplication while implementing the RA primitives which are necessary for developing Datalog applications. Our system is limited to a single GPU, and thus there are inherent scaling walls dictated by available VRAM on the GPU–by contrast, the largest unified nodes offer orders-of-magnitude more available RAM, supporting larger graphs. We view this work as a step towards multi-GPU joins across a cluster to develop a scalable backend for Datalog.

7 Acknowledgements

This work was funded in part by NSF RII Track-4 award 2132013, NSF collaborative research award 2217036, and NSF collaborative research award 2221811. 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.

8 Availability

The data, code, and documentation are all open-sourced and can be found at https://github.com/harp-lab/usenixATC23.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, January 2017.

[4] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[5] Martin Bravenboer and Yannis 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*, pages 243–262, 2009.

[6] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[7] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[8] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[9] Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702. Springer, 2012.

[10] Ke Fan, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. Exploring mpi collective i/o and file-per-process i/o for checkpointing a logical inference task. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 965–972, 2021.

[11] David Farrell. A simple gpu hash table, Mar 2020.

[12] Alex Fender, Brad Rees, and Joe Eaton. Rapids cugraph. In *Massive Graph Analytics*, pages 483–493. Chapman and Hall/CRC, 2022.

[13] Thomas Gilray and Sidharth Kumar. Toward parallel cfa with datalog, mpi, and cuda. In *Scheme and Functional Programming Workshop*, 2017.

[14] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 23–35, 2021.

[15] Oded Green, Zhihui Du, Sanyamee Patel, Zehui Xie, Hang Liu, and David A Bader. Anti-section transitive closure. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 192–201. IEEE, 2021.

[16] Chengxin Guo and Hong Chen. In-memory join algorithms on gpus for large-data. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1060–1067, 2019.

[17] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 881–884, 2014.

[18] Mark Harris. How to optimize data transfers in cuda c/c++, Dec 2012.

[19] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), December 2009.

[20] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 511–524, New York, NY, USA, 2008. Association for Computing Machinery.

[21] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216, 2011.

[22] Mark Jarzynski and Marc Olano. Hash functions for gpu rendering. *UMBC Computer Science and Electrical Engineering Department*, 2020.

[23] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[24] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, page 55–62, New York, NY, USA, 2012. Association for Computing Machinery.

[25] Sidharth Kumar and Thomas Gilray. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, volume 1, 2019.

[26] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In *International Conference on High Performance Computing*, pages 288–308. Springer, 2020.

[27] Argonne Leadership Computing Facility. Argonne leadership computing facility, 2022.

[28] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[29] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *International symposium on spatial and temporal databases*, pages 273–290. Springer, 2005.

[30] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on gpus: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550, 2015.

[31] Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient join algorithms for large database tables in a multi-gpu environment. *Proc. VLDB Endow.*, 14(4):708–720, December 2020.

[32] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013.

[34] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149, 2016.

[35] Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. Accelerating datalog applications with cudf. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 41–45, 2022.

[36] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709, 2019.

[37] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018.

[38] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1983.

[39] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, New York, NY, USA, 2014. ACM.