Optimizing Datalog for the GPU

Yihao Sun
ysun67@syr.edu
Syracuse University
Syracuse, New York, USA

Ahmedur Rahman Shovon
ashov@uic.edu
University of Illinois, Chicago
Chicago, Illinois, USA

Thomas Gilray
thomas.gilray@wsu.edu
Washington State University
Pullman, Washington, USA

Sidharth Kumar
sidharth@uic.edu
University of Illinois, Chicago
Chicago, Illinois, USA

Kristopher Micinski
kkmicins@syr.edu
Syracuse University
Syracuse, New York, USA

Abstract

Modern Datalog engines (*e.g.*, LogicBlox, Soufflé, `ddlog`) enable their users to write declarative queries which compute recursive deductions over extensional facts, leaving high-performance operationalization (query planning, semi-naïve evaluation, and parallelization) to the engine. Such engines form the backbone of modern high-throughput applications in static analysis, network monitoring, and social-media mining. In this paper, we present a methodology for implementing a modern in-memory Datalog engine on data center GPUs, allowing us to achieve significant (up to 45×) gains compared to Soufflé (a modern CPU-based engine) on context-sensitive points-to analysis of PostgreSQL. We present GPULog, a Datalog engine backend that implements iterated relational algebra kernels over a novel range-indexed data structure we call the hash-indexed sorted array (HISA). HISA combines the algorithmic benefits of incremental range-indexed relations with the raw computation throughput of operations over dense data structures. Our experiments show that GPULog is significantly faster than CPU-based Datalog engines while achieving a favorable memory footprint compared to contemporary GPU-based joins.

CCS Concepts: • **Computing methodologies → Parallel programming languages**.

Keywords: Datalog, GPU, Analytic Databases

1 Introduction

Declarative languages enable their users to write high-level logical rules that specify acceptable solutions to a problem, leaving efficient implementation to a high-performance backend. In particular, modern CPU-based Datalog engines power state-of-the-art systems in program analysis [5, 6, 12, 13], social-media mining [18, 44, 45, 49], and business analytics [41]. Such engines enable a user to specify deductions (forming an *intensional* database, henceforth IDB) over extensionally manifest relations (the *extensional* database, henceforth EDB). For example, given an input relation *Edge*(`from`,`to`), the program *REACH* computes *Edge*'s transitive closure:

$$Reach(\texttt{from}, \texttt{to}) \leftarrow Edge(\texttt{from}, \texttt{to}).$$
$$Reach(\texttt{from}, \texttt{to}) \leftarrow Edge(\texttt{from}, \texttt{mid}), Reach(\texttt{mid}, \texttt{to}).$$

The second rule requires recursive computation to a fixed-point: the engine iteratively discovers an ever-larger intensional database (*Reach*), starting from the extensional database (*Edge*). Unfortunately, best-in-class in-memory Datalogs hit scalability walls around 8–16 threads due to the challenges of working with locking, linked data structures in a parallel, shared-memory setting. For example, as we will soon see, when run at 32 threads on transitive closure, Soufflé [21] (a state-of-the-art CPU-based engine) spends 77.8% of its time in serialized tuple deduplication/insertion.

Compared to CPUs, GPUs offer hundreds of thousands of threads, along with extremely high memory throughput via HBM [20]. However, achieving optimal performance necessitates embracing the GPU's programming model. Traditionally, CPU-based graphics processing code has been accelerated via SIMD instructions (such as AVX and SSE), necessitating that all threads operate in lockstep and strongly penalizing thread divergence. While GPUs are inspired by this SIMD paradigm, modern GPGPU programming is SIMT in nature, allowing parallel execution at a per-thread granularity. Threads can diverge at the sub-warp level using advanced thread scheduling techniques [37]. As we will see, the SIMT nature of modern GPUs is a natural fit for Datalog, which stresses both massive data parallelism (*e.g.*, large joins), *and* task parallelism (*e.g.*, multiple rules).

In this paper, we introduce GPULog: a GPU-based library for parallel relational algebra kernels, enabling the execution of Datalog programs on modern GPUs. GPULog is backed by a novel SIMT data structure: the hash-indexed sorted array (HISA). We designed HISA to balance the concerns of modern Datalog on the GPU, enabling three critical tasks: (1) efficient range queries, necessary for joins, (2) lock-free deduplication, and (3) parallel iteration. Additionally, GPULog provides abstractions for optimized evaluation of n-way loop-joins we call *temporarily-materialized* joins, offering a space-for-time trade-off we found crucial to scale to high data loads.

We have used GPULog to implement and comprehensively study the performance of graph analytics (reachability, same generation) and program analysis (points-to analysis), evaluating GPULog against cuDF, GPUJoin, and Soufflé [21, 40, 48]. We observe improved (5×) join performance compared to state-of-the-art GPU join algorithms (due to efficient range queries enabled by HISA), along with significantly reduced memory footprint. Additionally, GPULog is the first-ever GPU Datalog implementation to achieve net-positive performance versus CPU-based engines, beating Soufflé (a leading CPU-based solver) by up to 45× (NVIDIA H100 GPULog vs. EPYC 7543P Soufflé). Our contributions are as follows:

- The Hash-Indexed Sorted Array (HISA), a novel relation-backing data structure which provides range querying while leveraging the efficiencies of dense representations.
- GPULog, a CUDA-based library for implementing Datalog queries on the GPU; GPULog uses HISA as its tuple representation. GPULog also leverages two novel strategies apropos Datalog on the GPU: eager buffer management and temporarily-materialized n-way joins.
- A thorough evaluation of GPULog, and a performance comparison between GPULog and state-of-the-art CPU and GPU-based Datalog engines and GPU joins; we show GPULog outperforms prior work by 5–45× with favorable memory footprint.

2 Datalog and Declarative Analytics

Datalog has yielded exciting results for a diverse set of data-heavy analytic applications such as static program analysis [7], graph mining [44], and machine learning [33]. Its growing popularity lies in its expressive and elegant semantics for specifying computations. Datalog programs comprise an extensional database (EDB) of explicit facts and an intensional database (IDB) of derived facts, transitively inferred from the rules and EDB [17]. The language's rules, written as Horn clauses, define relationships between data, and Datalog operates by iteratively applying these rules until no more knowledge is discovered [30]. Each rule consists of a head and a body. The general form of a Datalog rule is:

$$Head(...) \quad \leftarrow \quad Body_1(...), ..., Body_n(...).$$

The head, represented by a single predicate atom, signifies the derived fact to be inferred. The body, made up of comma-separated predicate atoms, specifies the conditions for the head's truth, with commas serving as a logical "AND" ($\wedge$). The implication symbol $\leftarrow$ connects the head and body, signaling that the head is derived when the body's conditions are met. This structure is closely related to concepts in relational database management systems (RDBMS). Notably, the sharing of logical variables across body clauses directly corresponds to a *join* operation on the related columns in an RDBMS. However, Datalog's foundation in logic programming transcends the capabilities of traditional SQL. It leverages this foundation to unlock greater expressive power, enabling functionalities like recursion and inference beyond simple data retrieval [43].

Semi-naïve evaluation. Modern Datalog engines owe their algorithmic benefits to incremental evaluation techniques such as semi-naïve evaluation [1], differential/timely dataflow [32, 34], and DBSP [8]. Following Soufflé, GPULog uses semi-naïve evaluation, which builds a *frontier* of freshly-discovered facts, avoiding the inevitable re-discovery of a fact at every subsequent iteration. To illustrate this process, consider the Same Generation (*SG*) query, used to determine if two nodes in a graph share a topological order:

$$SG(\mathsf{x}, \mathsf{y}) \quad \leftarrow \quad Edge(\mathsf{p}, \mathsf{x}), \; Edge(\mathsf{p}, \mathsf{y}), \; \mathsf{x} \neq \mathsf{y}.$$
$$SG(\mathsf{x}, \mathsf{y}) \quad \leftarrow \quad Edge(\mathsf{a}, \mathsf{x}), \; SG(\mathsf{a}, \mathsf{b}), \; Edge(\mathsf{b}, \mathsf{y}), \mathsf{x} \neq \mathsf{y}.$$

In this query, *Edge* represents extensional edges in the input graph. The first rule states that nodes x and y are in the same generation if they have a common parent node p. The second rule recursively derives *SG* by determining that x and y are in the same generation if there exists a node a with an outgoing edge to x, and another node b with an outgoing edge to y, such that a and b are themselves in the same generation.

The semi-naïve evaluation strategy optimizes this query execution, by maintaining the *SG* relation in three versions: "*new*" (containing tuples generated in the current iteration), "*delta*" (holding unique tuples added in the previous iteration), and "*full*" (comprising all tuples derived across all iterations). By performing the join operation solely on the *delta* relation, the engine significantly reduces redundant computation.

Example. Figure 1 illustrates each iteration of the *SG* query. During the first iteration, only the first rule generates new tuples since *SG* is empty and thus the second rule yields nothing. This rule applies when two edges in the graph merge from the same starting node but lead to different destination nodes; these edges are joined to generate new *SG* tuples. For instance, the tuple $SG(7, 8)$ is produced by joining $Edge(4, 7)$ and $Edge(4, 8)$, both of which originate from node 4. Upon completing the first iteration, the newly generated tuples are moved into both the *delta* and *full* versions of the SG relation, preparing them for subsequent computation.
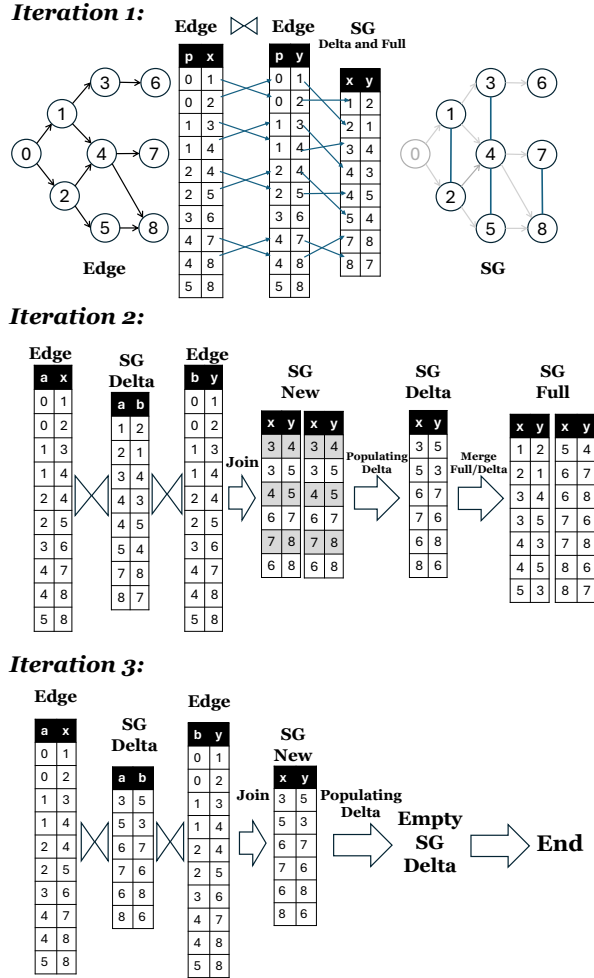
Figure 1. The execution results of each iteration in Same Generation (SG) query.

The middle of Figure 1 shows the second iteration involving a three-way join: $Edge \bowtie SG_{delta} \bowtie Edge$, leading to the derivation of indirect same generation tuples, such as $SG(6, 8)$, from $Edge(3, 6)$, $Edge(4, 8)$, and $SG(3, 4)$ (in the *delta* version). Notice that the *new* version of the SG relation, generated post-join, contains duplicates tuples from its *full* version, such as $SG(3, 5)$ and $SG(7, 8)$. To ensure maintain the invariant that *delta* and *full* are disjoint, a deduplication process is implemented both within the *delta* and against the *full* set as the *new* tuples are allocated.

After completing the second iteration, the Datalog engine proceeds with the third iteration, as shown in the bottom of Figure 1. The previous iteration's SG_{delta} is used as the input for the join operations in the current iteration. The join results in new tuples SG_{new}; however, all of these tuples are already present in SG_{full}, leading to an empty SG_{delta}. This indicates that the query has reached its fixpoint. We revisit

semi-naïve evaluation in more detail within the context of our GPU-based implementation in Section 4.

3 GPU Datalog: Challenges and Concerns

The high performance of modern Datalog engines (such as Soufflé [21] and BigDatalog [45]) is due to a combination of factors, including compilation, data structures, and semi-naïve evaluation. Among these, we focus on the design of a GPU-based relation-backing data structure. In the context of relevant related work, we outline four key requirements demanded of a GPU-based relation-backing data structure. No state-of-the-art implementations sufficiently satisfy all of our requirements (at least, in a fully-general manner).

[R1]: Efficient Range-Querying. Joins $R(\ldots) \bowtie Q(\ldots)$ are operationalized in modern Datalog engines via loop-joins, scanning R and range-querying Q, or mutatis mutandis for Q and R. To support range-querying, tuples are organized (*e.g.*, via indexing) to associate sets of tuples with a set of join columns. We call the scanned relation the *outer* relation, and the range-indexed relation the *inner* relation. Compared to traditional RDBMS systems—which cannot possibly compute optimal indices due to the ad-hoc nature of querying—modern Datalog engines perform indexing for *every* query [50]. Modern engines employ a mix of minimally-locking linked data structures such as B-trees and Bries [22, 23, 42], and are carefully tuned for optimal performance on shared-memory, CPU-based systems.

We can now identify the first requirement for our GPU-based data structure: it must *support fast range queries [R1]*. Several prior GPU-based approaches (*e.g.*, GPUJoin) have implemented SIMT-friendly data structures to enable fast range queries. For example, relations may be stored in hashmaps whose indexed columns serve as keys, and whose non-indexed columns act as values. By leveraging atomic operations for key updating and open addressing, these hashmaps may be constructed in a data-parallel manner.

[R2]: Parallel Iteration. Hash-based data structures, due to their sparse nature, face limitations in efficient iteration. To enable parallel iteration on the GPU, the data stored in these hash-based structures must undergo a serialization process. This involves traversing the hashmaps and copying all the elements into a contiguous array. Consequently, while these data structures are suitable for inner relations where range query speed is the focus, they are not ideal for outer relations that require iteration. It can be argued that keeping the outer relation as an array and the inner as a hash table could resolve this issue. However, in Datalog, it is common for a relation to function both as an inner and outer relation. As an example, the first rule of the same generation query in Section 2, involves a self-join on the Edge relation.

The limitations of hash-based data structures, particularly their iterability, make them less favorable for outer relations
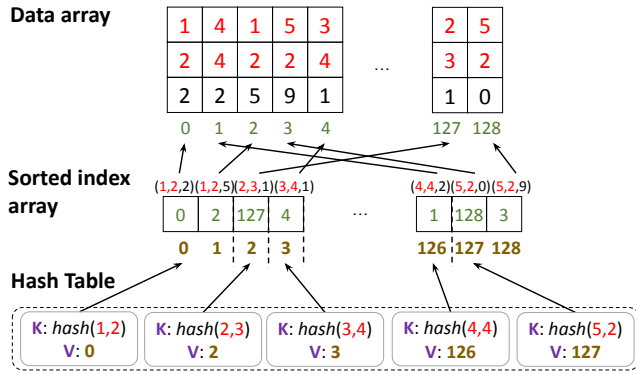
Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Sidharth Kumar, & Kristopher Micinski

Figure 2. Example of a 3-arity relation with 2 join-columns (shown in color red) stored in HISA.

in join operations where GPU-parallelizable traversal is crucial. Thus, our second requirement for the data structure is to support parallel iteration [R2].

[R3]: Multiple Join Columns. Real-world analysis queries can contain multiple join columns. For instance, consider the query from DDisasm [13] (a Datalog-based disassembler):

$value\_reg\_unsupported(\text{EA}, \text{Reg}) \quad \leftarrow$
$\qquad def\_used.for\_address(\text{EA}, \text{Reg}, \_),$
$\qquad arch.memory\_access(\text{ LOAD}, \text{EA},$
$\qquad\qquad\qquad\qquad \_, \_, \text{Reg}, \text{RegBase}, \_, \_, \_),$
$\qquad RegBase \neq \text{NONE} \qquad .$

This *join* operation involving the *def_used.for_address* and *arch.memory_access* is executed on two columns: EA and Reg. Parallel construction of hashmaps on GPUs depends on atomic operations, which have a size limitation of 64 bits, or at most, 128 bits. As a result, directly using multiple columns as keys in the hashmap can be problematic, especially when the combined size of these columns exceeds the 128-bit limit. This restriction poses challenges in efficiently storing and accessing data with multiple columns in GPU-based hashmap implementations. This brings us to the third requirement of our data structure, it must *support multiple join columns [R3]*.

[R4]: Efficient deduplication. As a final requirement, the data structure should *support efficient deduplication* [R4]. As mentioned earlier, deduplication is a crucial part of semi-naïve evaluation in modern Datalog engines (see Figure 1). Unfortunately, many existing GPU Datalogs do not support deduplication (RedFox [55] and GPUDatalog [31]).

4 The Hash-Indexed Sorted Array (HISA)

To meet all four requirements ([R1], [R2], [R3], and [R4]), we developed a novel data structure we call the Hash-Indexed Sorted Array (HISA). HISA, inspired by *HashGraph* [15], is a custom-designed data structure consisting of three interconnected layers. At its foundation lies the data array, which stores the original tuple data. Built upon this data array is a

sorted index array, which stores the indices of the tuples in ascending order. The third layer is an open-addressing hash table, constructed on top of the sorted index array. HISA is different from other data structures as instead of building the hash table directly on the actual data (tuples), we construct it on the sorted index array. This approach allows us to store references only for tuples with unique join column keys, resulting in a more efficient use of memory.

Figure 2 sketches an example of a HISA data structure at runtime (in VRAM). We will now walk through each of the tiers, and discuss how they collectively enable [R1]–[R4].

4.1 Data Array

The *data array* represents the original GPU buffer that stores all the tuples of a relation transferred from the CPU to the GPU. The k-ary relation data can be seen as a 2D array of size $n \times k$, where n is the total number of tuples/rows and k is the number of columns of the relation. An example of this 2D array can be seen at the top of Figure 2. Here $n = 129$ and $k = 3$, but, for the sake of clarity in exposition, the 2D array is shown in its transposed view. We store this 2D data in row-major order in the GPU buffer. Therefore, the tuples of Figure 2 are stored in a buffer that looks like – $\{1, 2, 2, 4, 4, 2, 1, 2, 5 \ldots, 2, 3, 1, 5, 2, 0\}$.

This densely-packed contiguous arrangement, in contrast to sparse data structures, simplifies access for the GPU. GPU threads can then efficiently perform strided access to underlying the data array.

This simple layout makes it easy to implement parallel data retrieval, addressing the requirement for fast iteration [R2]. Coalesced memory access, enabled by our layout, allows multiple GPU threads to fetch data from the same memory block simultaneously, improving cache performance and optimizing memory operations. We provide implementation details of this parallel data access in Section 5.1, in the context of iterating the outer relation for a join operation.

Merge operation. Merging two HISAs typically occurs when combining the full and delta versions of a relation. In semi-naïve evaluation, to avoid redundant computation, the delta relation contains only fresh and unique tuples generated in the previous iteration, deduplicated against the full relation. Consequently, when merging delta and full, no additional deduplication is needed—the delta's data array can be directly concatenated to the full's data array.

4.2 Sorted Index Array

The sorted index array stores the indices of the tuples from the data array in ascending order. The indices are arranged based on a lexicographical ordering of the tuples. This ensures the join-columns appear first, followed by the remaining columns. For example, consider a 3-arity data array with tuples $\{2, 1, 5\}$, $\{2, 5, 9\}$, and $\{2, 1, 2\}$, where the second column is the join-column. The corresponding sorted index

---

Algorithm 1 Construction of Sorted Index Array H_i for Relation H

1: **for** all tuple T in H.data_array **parallel do**
2:     $jc \leftarrow$ [join columns of T]
3:     $njc \leftarrow$ [non-join columns of T]
4:     Insert $(jc ++ njc)$ into H_i.data_array
5: **end for**
6: H_i.sorted_index_array $\leftarrow [0 \dots H$.tuple_size]
7: **for** i in $[0 \dots H$.arity] **do**
8:     $col_{tmp} \leftarrow$ the i_{th} of column of tuples in H_i
9:     stable sort H_i.sorted_index_array use col_{tmp} as key
10: **end for**

---

array would be $1, 0, 2$, because the tuples, after reordering the join column to index 0, follow the lexicographic ordering $(1, 2, 2) < (1, 2, 5) < (5, 2, 9)$. In essence, the sorted index array decouples the sorting order of tuples from their physical arrangement in the data array. It maintains the positions of tuples within the data array, and these positions are sorted based on the lexicographical order of their associated tuples.

Construction. We extensively use NVIDIA's Thrust library [36] to perform tasks such as copying, gathering, and sorting. The first step in the process of creating the sorted index array is to reorder the columns, which we perform using Thrust's transform function. This can be seen in line numbers 1 to 5 in the Algorithm 1. After the reordering phase, we populate the sorted index array, by using Thrust's stable sort utility. This phase performs a stable sort based on the least significant column (rightmost) of the tuple and progresses towards sorting by the most significant column (leftmost). This sorting process is similar to radix sort; however, in this case, tuples are sorted one column at a time (rather than one bit at a time). This can be seen in line numbers 7 through 10.

Fast range-queries. Rearranging the column order to position the join columns at the beginning and then sorting the tuples offers two significant benefits. First, it enables efficient range queries, and second, it facilitates the fast merging of two relations. While fast-range queries are crucial for performing join operations, merging relations is an essential part of semi-naïve evaluation. As discussed in Section 2, the new relation is merged into the full relation at each iteration of the fixed point.

The sorted-index array works in tandem with the hashmap (described in the next subsection) to execute range queries. The hash map directly positions a GPU thread (in O(1) time) at the appropriate location within the sorted index array. Since the sorting process groups together tuples with identical join-column values, a range query can easily retrieve the entire set of tuples sharing the same join-column values by a linear scan.

Deduplication. HISA supports deduplication [R4] by first sorting all columns in lexical order, facilitating the rapid

---

Algorithm 2 Hash table construction algorithm

1: **for** each tuple T in input relation, perform in parallel **do**
2:     $H \leftarrow hash(T)$
3:     $I \leftarrow H$ (mod hash_table_size)
4:     $T_{pos} \leftarrow$ get_position(H, sorted_index_array)
5:     **while** T is not inserted **do**
6:         $K, V \leftarrow$ hash_table[I]
7:         $K' \leftarrow$ AtomicCAS(K, H)
8:         **if** $K' = H$ or (K, V) is uninitialized **then**
9:             **while** $T_{pos} < V$ **do**
10:                 $V \leftarrow$ AtomicCAS(V, T_{pos})
11:                 **if** $V = T_{pos}$ **then**
12:                     **break**   ▷ Inserted T in *hash_table*
13:                 **end if**
14:             **end while**
15:         **else**
16:             Perform linear probing on I
17:         **end if**
18:     **end while**
19: **end for**

---

identification and removal of duplicates. The deduplication process is then executed by comparing each tuple with its adjacent tuple during a parallel scan. This approach ensures both efficient grouping for joins and effective deduplication, leveraging the parallel processing capabilities of the GPU.

Merge operation. Parallel merging of two sorted arrays is a common pattern in GPGPU programming. We utilize the path merge algorithm [16] provided by NVIDIA's Thrust library to merge the sorted index arrays of two HISAs.

4.3 Open-Addressing Hash Table

The sorted index array stores tuples in increasing order, clustering tuples which share the same join-columns. To further optimize range queries, enabling retrieval of all tuples sharing the same join columns and meeting requirement [R2], HISA incorporates an open-addressing-based hash table. Using a hash table avoids control flow divergence and improves memory access efficiency compared to other range query techniques, such as tree-based searches.

The hash table is constructed to store distinct hash values (as keys) derived from the tuples of the data array, while only considering the join-columns to compute the hash. These keys are then associated with the smallest index of a tuple in the sorted index array (as values) that contains the corresponding join-column values. An example of this hash table can be seen at the bottom of Figure 2. In this example, the hash table entry on the far right captures the hash value of join-columns $(5, 2)$ and associates it with the starting position of all tuples having the join-column $(5, 2)$, indicated by the 127th position in the sorted index array. Instead of directly storing the join column values—which may exceed 64 bits in size—we opt to store their hash value as the key; this strategy effectively meets requirement [R3].

Construction. The pseudo-code for the construction of the hash table can be seen in Algorithm 2. We construct an open-addressing hash table, employing linear probing to handle collisions. The keys of the hash table are generated from the data array, while the values are derived from the sorted index array. GPU threads go through the sorted index array, which allows them to access all elements of the data array. They calculate hash values for every tuple, using only the join-column values and use these hash values as keys in the hash table. The hash table's values store the positions of the corresponding keys in the sorted index array, enabling quick lookup and access to the relevant data elements. To handle collisions that occur when two keys map to the same slot in the hash table, we employ linear probing. However, since insertion is performed in parallel by thousands of threads, it is possible that during collision resolution, two threads with hash collisions attempt to write to the same hash slot. To address this challenge, we utilize the `atomicCAS` (Compare-and-Swap) operator at line 7 of the algorithm, which ensures that only one thread can successfully update a slot at a time.

In parallel execution, multiple GPU threads update smallest index of a tuple in the sorted index array simultaneously. This also leads to a race condition when two tuples sharing identical join column values are concurrently inserted into the hash table by two separate GPU threads. We handle this from line 9 to line 14 of the algorithm, also using `atomicCAS` operator, which updates a value only if the new key-value pair is smaller than one existing in the hash table.

To conduct a range query, the process begins with calculating the hash value of the join columns. Based on this hash value, the method navigates to the smallest tuple possessing the specified join-column values and then proceeds with a linear scan until encountering a tuple where the join-column value no longer matches. We show this in more detail in the context of a join operation in Section V.A.

Merge operation. Currently, hash table merging of HISA is done by inserting all elements from the Delta hash table into the Full table. For each existing key, the value is updated if the Delta value is smaller than the current value in the Full hash table. While we consider more efficient algorithms (e.g., Cuckoo Hashing[29]) as an important future direction, GPU hash tables can process up to 0.5 billion keys per second, making this approach faster than merging sorted arrays, which requires significant time for buffer allocation and deallocation during path-merging.

5 GPUʟᴏɢ: GPU-powered datalog engine

In this section, we introduce GPUʟᴏɢ, our GPU-accelerated implementation of Datalog that leverages the HISA to optimize performance and efficiently utilize the GPU. We focus our discussion on three distinguishing aspects: (a) implementation of the semi-naïve evaluation in the fixed-point loop, (b) a novel optimization to significantly accelerate *n*-way
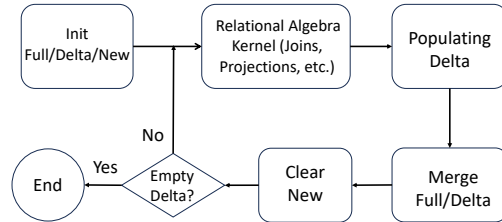

Figure 3. Workflow of semi-naïve evaluation.

joins: temporary materialization, and (c) eager buffer management, which leverages amortization to improve memory allocation.

5.1 Evaluation of Datalog Programs

Datalog's evaluation is often given in terms of relational algebra, which includes relational operators such as join and projection. Evaluation executes these kernels in a loop. In each iteration of the loop, compute kernels (compiled from the program rules) generate new tuples (facts); these newly-discovered facts may then trigger further deduction in subsequent iterations. The process continues until a fixed point is reached, where no new facts can be derived. Figure 3 illustrates the detailed steps involved in this evaluation process, namely "Join," "Populate Delta," "Merge Full/Delta," and "Clear New."

Implementing Joins. The bulk of the evaluation overhead of modern Datalog evaluation lies in iterated joins. For example, consider the following query:

$$Foobar(c, d) \quad \leftarrow \quad Foo(a, b, c), Bar(a, b, d).$$

This join operation involves two relations, each with three attributes, sharing the first two columns as join columns in Figure 4. To facilitate parallel iteration over all tuples of the outer relation, we use the data array component of the HISA data structure. As shown in the figure, we access the data array of the outer relation in *stride* units. The size of each stride corresponds to the total number of threads employed for concurrent execution. Each GPU thread concurrently retrieves a tuple within a stride, where the thread ID corresponds to the tuple's offset within that stride. In the example shown all tuples of the outer relation are accessed in two strides, the first stride ranges from tuples $Foo(2, 3, 5)$ to $Foo(5, 2, 4)$ and the second stride ranges from $Foo(2, 3, 2)$ to $Foo(5, 2, 6)$. We encapsulate this strided access in the form of a parallel for in line number 1 of the Algorithm 3. Users can configure the stride size based on their GPU, with a recommended size being 32 times the number of stream processors. By accessing the data in the outer relation using this stride-based approach, data locality is improved, leading to optimal utilization of the cache and enhancing overall performance.
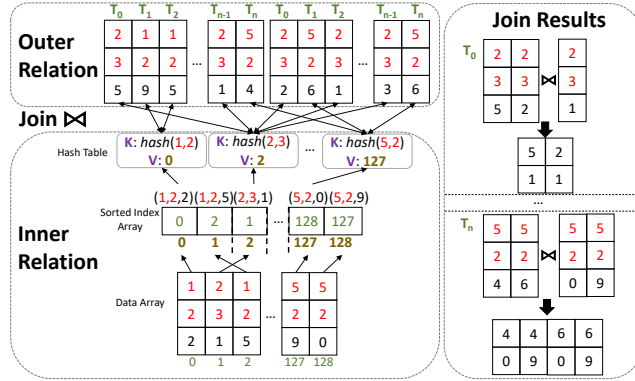
Figure 4. Example of a join between two 3-arity relations *Bar* (inner) and *Foo* (outer).

The lower section of the figure demonstrates how each thread queries the inner relation. The range queries are facilitated by using the hash table and the sorted index array components of HISA. The hash table is used to find the starting position of the relevant queries in the sorted index array (lines 2 to 5), which is then linearly scanned to retrieve all matching tuples from the data array (lines 6 to 10). For instance, thread T_0 operates on the outer relation tuple $Foo(2, 3, 5)$ and $Foo(2, 3, 2)$. After hashing column $(2, 3)$, GPULOG uses the hash table of the HISA structure to find the smallest tuple at position 2 in the *Bar* relation's sorted index array, yielding $Bar(2, 3, 1)$. A scan starting from this position reveals that the only matching tuple in the *Bar* relation is $Bar(2, 3, 1)$. On the right side of the figure, the join results show that tuples *Foobar(5,1)* and *Foobar(2,1)* are generated as a result of the join between matched tuples from the outer and inner relations.

Although both employ hash join-like algorithms, GPU-LOG differs from the join algorithm used in another Datalog engine prototype, GPUJoin [48]. In GPUJoin, the hash table directly contains all tuples, and during the join, tuples are accessed through linear probing of the hash table. This design can lead to larger hash tables and increased memory overhead, especially when a low load factor is used for faster construction. In contrast, our tool's join algorithm accesses the hash table only to find the starting position of tuples sharing the same indexed column. This approach keeps the hash table size small and allows for fast construction.

Populating delta. The next step in the fixpoint loop involves populating the *delta* relation, which will be used in the join phase of the next iteration. In this step, *delta* is constructed by removing from *new* the tuples that are already present in *full*. GPULOG accomplishes this using a relational algebra "set difference" applied to the *new* and *full* relations. It is worth noting that some prior work, such as GPUJoin, fuses this step with the merging step by directly merging the non-deduplicated *delta* relation and then deduplicating the

---

Algorithm 3 Parallel Binary Join on GPU

1: **for** T_{outer} in $Relation_{outer}$ **parallel do**
2:     $ht \leftarrow \text{hash}(T_{outer}.\text{join\_columns})$
3:     $index\_pos \leftarrow ht \% \text{ hash table size of } Relation_{inner}$
4:     $pos \leftarrow Relation_{inner}.\text{sorted\_index\_array}[index\_pos]$
5:     $T_{inner} \leftarrow \text{tuple at } pos \text{ in } Relation_{inner}.\text{data\_array}$
6:     **while** $T_{inner}.\text{join\_columns} \neq T_{outer}.\text{join\_columns}$ **do**
7:         generate result tuples based on T_{outer} and T_{inner}
8:         pos++
9:         $T_{inner} \leftarrow \text{tuple at } pos$
10:     **end while**
11: **end for**

---

merged *full* relation. This fusion approach is efficient when the size of the *full* relation is small. However, deduplicating the *full* relation requires a full scan of all the tuples in the *full* relation, which becomes highly expensive as the size of the *full* relation grows. Therefore, GPULOG takes a different approach by separating the *delta* relation population into a distinct phase.

Merging Full with Delta and Clearing New. The final step before fixpoint checking involves merging all the tuples within the *delta* relation into the *full* relation, followed by the removal of all tuples within the *new* relation. Leveraging the HISA bulk insertion technique outlined in Section 4.2, we incorporate all deduplicated new tuples from *delta* into *full* in parallel on the GPU. It is worth noting that the management of buffer memory during merging between *full* and *delta* plays a crucial role in ensuring the optimal performance of larger queries.

5.2 Temporarily-Materialized *n*-way Joins

While simple Datalog queries use only binary joins, practical applications include *n*-way joins. An example is the Same Generation (*SG*) query mentioned in Section 2. The second rule of *SG* represents an inductive case that necessitates joining *SG* with *Edge* twice. When employing semi-naïve evaluation, its join plan may be expressed via the following relational algebra operation:

$$SG_{new} \quad = \quad Edge_{full} \bowtie SG_{delta} \bowtie Edge_{full}$$

Various mechanisms exist for evaluating *n*-way joins; while special-purpose joins (such Leapfrog Triejoin [52]) can achieve worst-case optimal bounds, they do so by imposing representation restrictions. Modern engines tend to favor nested-loop joins [21], in which case *n*-way joins are ordered into sequences (or trees) of binary joins, with an emphasis on optimizing the join order[57]. In the case of *SG*, it first loops over Edge_{full} and within the loop body, it further loops on all matched tuples in Edge_{delta}. This generates another nested loop over to join with Edge_{full}.

Due to the architecture of GPU, conditional branching in the program can lead to threads idling during an *n*-way join.
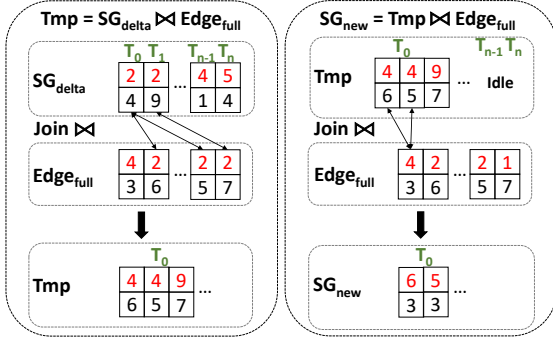
Figure 5. Thread waiting in nested non-materialized join due to workload imbalance. T_0 to T_n are in the same warp.

For example, in Figure 5, thread T_0 is assigned to compute the join result for the tuple $SG(2, 4)$. During the first join with $Edge_{full}$, it generates numerous temporary results. Unfortunately, thread T_{n-1} and thread T_n lie within the same GPU warp and do not find any matching tuples. In the subsequent join between the first set of join results and $Edge_{full}$, T_0 must continue the computation, while threads T_{n-1} and T_n, have to remain idle until T_0 completes the second join and returns.

Another approach to divide the workload among threads involves materializing each sub-join. We leverage this observation by using a "temporary materialized" buffer. In SG, the second rule can be seen as two sequential binary joins:

$$
\begin{aligned}
Tmp(b, x) &\leftarrow Edge(a, x),\ SG(a, b). \\
SG(x, y) &\leftarrow Edge(b, y),\ Tmp(b, x).
\end{aligned}
$$

In relational algebra, an explicit join plan may be written as:

$$
\begin{aligned}
Tmp &= SG_{delta} \bowtie Edge_{full} \\
SG_{new} &= Tmp \bowtie Edge_{full}
\end{aligned}
$$

This split does not make a significant impact on performance for CPU based system. However, in GPU-based systems, having two sequential joins scheduled into two separate executions of GPU kernel functions is notably faster from the first method, where only one GPU kernel function executes. By utilizing the technique of temporal materialization, data parallelism is optimized to its fullest extent. This optimization ensures that all available threads are actively engaged in computation, effectively eliminating any idle time.

The first join's workload is based on tuples in SG_{delta}. By contrast, the workload of the second join is divided based on the Tmp tuples. In the previous example, temporary join result tuples like $Tmp(9, 7)$ are assigned to other idle threads, such as thread T_{n-1}, T_n, rather than all being processed by the busy thread T_0. In this way, all threads in the same GPU wrap will have a similar workload, eliminating idle threads.

5.3 Buffer Management and Amortized Allocation

As we will see in Section 6.3, the merge operation of *delta* tuples into the *full* version of the relation becomes a bottleneck, consuming up to 45% of the total execution time. Buffer management during the merge process is particularly expensive, as it requires reallocating new buffers and copying data equivalent to the combined size of the *full* relation and the *delta* for each iteration. To address this, GPULOG employs a strategy called Eager Buffer Management (EBM). Unlike the buffer management policies in other GPU Datalog engines, such as GPUJoin, the buffer allocated using EBM in GPULOG is not freed immediately after the merge operation. Before performing the merge, GPULOG checks if the buffer allocated in the previous iteration is sufficient for the current merge operation. If it is, the buffer is reused; if not, rather than allocating a buffer with the exact size of the full and delta tuples, GPULOG allocates a buffer sized to the full tuple size plus k times the delta tuple size, where k is a tunable parameter depending on the total size of VRAM. This strategy reduces the overhead of frequent buffer allocations in every iteration and uses a small amount of extra memory to enable faster iterative computation.

6 Evaluation

In this section, we conduct a comprehensive performance evaluation of GPULOG, demonstrating the impact of our optimization techniques. Our assessment begins by analyzing how eager buffer management influences both query execution time and system memory usage. We then compare the performance of GPULOG against existing Datalog engines, utilizing well-established Datalog queries and real-world datasets. Due to the limited availability of mature GPU-based Datalog engines and the inaccessibility of GPUDatalog [31] in both source and binary forms, we also include a state-of-the-art CPU-based system (Soufflé) in our comparison. Although DCDatalog [56] recently claims to have surpassed Soufflé in speed, the paper's artifacts are not accessible for evaluation. Finally, we demonstrate the practicality of GPU-LOG for program analysis, showing significant performance boosts over CPU-based solutions.

6.1 Environment Setup

GPULOG and Other GPU-Based Systems. Except for Section 6.6, all our CUDA-related code runs on a server equipped with a 64 core AMD EPYC 7713 CPU (code name Milan), and an NVIDIA H100 PCIe 80GB GPU. The CUDA toolkit version we use is 11.8. For cuDF, we employ version 23.10, the latest version available at the time of our evaluation, also running under CUDA 11.8. The code for Datalog queries in cuDF and GPUJoin is sourced from its public GitHub repository [46]. In Section 6.6, our GPULOG-HIP implementation is tested on 2 machines. One has dual AMD EPYC 7713 processors together with four 64GB AMD Instinct MI250 cards. Another

has dual AMD EPYC 7742 and four 32GB AMD MI50 GPUs. In our benchmark, only one GPU for each system is used. The NVIDIA A100 results are tested on a machine with a 32-core Intel Xeon Gold 6338 CPU. Although the testbed we rented utilizes high-end CPUs, it is important to note that using lower-performance CPUs does not affect our GPU tool's performance. We validated our A100 results on a less powerful CPU and obtained nearly identical results.

CPU Based System: Soufflé. Our Soufflé experiments are conducted on a single node (1×AMD EPYC 7543P) with 512G RAM. All Soufflé queries are compiled into C++ code and then compiled using gcc-10 with the optimization flag -O3. To fully utilize all 32 cores on the test platform's CPU during compilation, we used the -j 32 option with the build system . We found this setting delivered the best runtime on our test platform.

6.2 Test Programs and Datasets

We evaluate reachability (*REACH*, Section I), same generation (*SG*, Section II), and context-sensitive program analysis (*CSPA*) [26]. We include *REACH* as a common baseline which stresses iterated binary joins, without any need for temporal materialization. We include *SG* and *CSPA* as queries which require more intelligence in join planning and stresses support for *n*-way joins. Recall the code for *SG* was introduced in Section 5.2. We ran our experiments using ten large graphs for both *REACH* and *SG*. These graphs were chosen from the Stanford Network Analysis Project network dataset, SuiteSparse, and road network datasets [9, 27, 28], and vary in size and application domain. For *CSPA*, we used input data provided by the authors of RecStep [12].

6.3 Evaluating Eager Buffer Management

We assess the efficacy of our Eager Buffer Management (EBM) mechanism by conducting a comparative analysis of runtime and memory usage with EBM enabled and disabled, using the *REACH* query on five distinct large graphs. Our results are presented in Table 1. This table is organized into four groups of columns. The first group specifies the datasets used in the experiment. The second group of columns provides insights into the iteration counts within the *REACH* query. This group is further divided into two sub-columns. "Total" denotes the total number of iterations, while "Tail" refers to the count of tail iterations, where the number of *delta* tuples generated in that iteration is less than 1% of the total tuples in the resulting *Reach* relation. Notably, the absence of a tail iteration number for the "usroads" dataset is due to the fact that the number of *delta* tuples generated in every iteration is less than 1%. The third group of columns displays the running time and speed-up ratio achieved by enabling Eager Buffer Management (EBM). It reveals that EBM accelerates all *REACH* queries, with greater benefits in

Table 1. Comparing runtime and memory usage of *REACH* in GPULog with and without eager buffer management on NVIDIA H100

	Iterations		Query Time (s)		Memory(GB)	
Dataset	Total	Tail	Normal	Eager	Normal	Eager
usroads	606	/	52.42	17.53	20.35	26.84
vsp_finan	520	491	59.08	21.91	20.22	28.26
fe_ocean	247	90	47.19	23.36	37.97	50.43
com-dblp	31	18	17.83	14.30	43.24	60.18
Gnutella31	31	17	4.80	3.76	20.22	28.26

longer iterations, especially long tail ones. "usroads" benefits the most, achieving a 3× acceleration. This is because relations that have a higher percentage of tail iterations are bound to benefit more from EBM, as the extra buffer allocated as part of the EBM will be able to accommodate the buffer merge requirements of a large fraction of the tail iterations, requiring fewer memory reallocations.

Table 2. Reachability execution time comparison: GPULog (NVIDIA H100) vs. Soufflé (AMD Milan CPU 32 cores), GPUJoin, and cuDF on large graphs (OOM: out of memory).

Dataset name	*Reach* edges	Time (s)			
		GPULog	Soufflé	GPUJoin	cuDF
com-dblp	1.91B	**14.30**	232.99	OOM	OOM
fe_ocean	1.67B	**23.36**	292.15	100.30	OOM
vsp_finan	910M	**21.91**	239.33	125.94	OOM
Gnutella31	884M	**5.58**	96.82	OOM	OOM
fe_body	156M	**3.76**	23.40	22.35	OOM
SF.cedge	80M	**1.63**	33.27	3.76	64.29

6.4 GPULog vs. SOTA GPU joins

In this section, we evaluate the performance of the *REACH* query using GPULog, GPUJoin, and cuDF, with Soufflé serving as the baseline. The results are summarized in Table 2. The datasets used in the experiments represent a diverse array of sources, ensuring a comprehensive assessment that spans scientific communities, P2P networks, random graphs, and road networks. This dataset diversity helps mitigate any bias towards specific graph categories.

GPULog consistently emerges as the fastest engine in all test cases. Compared to GPUJoin, GPULog exhibits a remarkable speedup, with ratios often exceeding 3x. In the "fe_body" testcase, the speedup ratio reaches an impressive 6x. Notably, GPUJoin, designed specifically for reachability queries, stores output relations in raw arrays. This specialization suggests that GPULog could achieve even higher speedup ratios when the method of GPUJoin is applied to general queries. Compared to GPUJoin, GPULog used less memory: GPUJoin OOM'd twice during testing. This is because of its of use

Table 3. Same Generation (*SG*) execution time comparison: GPU-LOG vs. Soufflé and cuDF. Soufflé running 32 core AMD Milan CPU; GPULOG and cuDF running on NVIDIA H100 GPU.

Dataset	SG size	Time (s)			
		GPULOG	HIP	Soufflé	cuDF
fe_body	408M	**5.05**	19.57	74.26	OOM
loc-Brightkite	92.3M	**3.42**	14.00	48.18	OOM
fe_sphere	205M	**2.36**	8.48	48.12	OOM
SF.cedges	382M	**5.54**	20.57	68.88	OOM
CA-HepTH	74M	**2.79**	5.92	20.12	21.24
ego-Facebook	15M	**1.23**	2.81	17.01	19.07

open-addressing-based hashmaps for storing relations, necessitating a low hashtable loading factor to facilitate joins. By contrast, HISA employs hash tables only for accelerating range fetching on indexed relations. This results in efficient access even with a higher load factor (0.8).

We present another test case, *SG*, to further validate our previous discussions. The results are presented in Table 3. The *SG* query involves *n*-way joins and represents more general use cases. As GPUJoin does not support the SG query, we exclusively compare GPULOG with cuDF. Across all six test cases, cuDF encounters out-of-memory (OOM) errors four times, while the non-OOM test cases, although approaching the baseline performance of Soufflé, remain considerably slower than GPULOG. GPULOG consistently delivers stable performance, running nearly 7× faster than the baseline in most of cases, while not experiencing any OOM error.

6.5 Context-Sensitive Program Analysis

Program analysis is one of the most popular applications of Datalog, with several state-of-the-art tools (DOOP [6], cclyzer [5], and ddisasm [13]) implemented using Soufflé Datalog. To evaluate GPULOG's potential application to program analysis, we implemented context-sensitive points-to analysis (*CSPA*), reproducing the experiments of Graspan [53]. We elide the full query for space, it includes rules which initialize and propagate value flows, while simultaneously discovering alias information; Graspan uses method cloning (effecting inlining) for context sensitivity. Using Graspan's data, we ran *CSPA* on httpd, (a statically-linked subset of) linux, and postgres [53]. The results of our evaluation are presented in Table 4; the table lists the size of input and output relations (second and third columns) and total running time (fourth column). All relation sizes match that of Souffé's.

We see a roughly 35-45× speedup versus Soufflé under ideal conditions (Soufflé use 32 cores CPU with all optimizations, GPULOG running on NVIDIA H100 with maximum buffer size), demonstrating the significant performance advantage of GPULOG over the CPU-based Soufflé. This is

mainly because the memory-heavy nature of program analysis workloads. Memory bandwidth and throughput significantly affect performance, and CPU systems usually have lower memory bandwidth. For example, the AMD Milan's bandwidth is around 190 GB/s, while a data center GPU like the NVIDIA H100 can have up to 3.35 TB/s. The lower memory speed and throughput prevent CPU-based systems from fully utilizing their high single-core performance. Our detailed analysis on CPU utilization further proves this, despite utilizing all 32 CPU cores during computation, the efficiency of Soufflé was relatively low, with CPU ratios of 571%, 449%, and 683% for the httpd, Linux, and PostgreSQL datasets, respectively. Note that a 100% utilization across 32 cores should yield a CPU ratio of 3200%.

Figure 6 breaks down different phases of GPULOG when running our queries. In this figure, "Deduplication" shows time spent coalescing previously-discovered facts. "Indexing Full/Delta" represents the time required for creating hash indexing within GPULOG's hybrid data structure. ''Merge Delta/Full" captures the time spent on inserting all tuples from the delta relation into the full relation using GPU merge. Finally, "Join" represents the actual time spent performing join operations across all relations. The join (39% of total runtime) and merge (42% of total runtime) operations stand out as the most time-consuming phases in our pipeline, as depicted in the red and blue bar in the Figure 6. Since performing joins between relations is computationally intensive and one of our main focuses, we expect this phase to occupy a significant portion of the total runtime. By leveraging the HISA datastructure, GPULOG is able to make these joins more efficient than previous GPU-based systems [47, 48]. However, as the figure indicates, the merge operation is even more time-consuming than the join phase. This is due to the reason that lock-free style parallel path merge aglorithm we used require pre-allocating a result buffer that matches the combined size as total size of delta and full. Since the full relation in Datalog is typically large, this result buffer can be substantial. Allocating and freeing such a large buffer in every iteration is expensive, and, according to our detailed timing results, this step becomes even more time-consuming than the join and indexing operations in the query. To partially address this challenge, we introduce an amortized buffer management scheme (described in Section 5.3). While this approach doesn't fully resolve the issue, it mitigates the need to allocate buffers at every iteration, thereby reducing the overall computational load.

6.6 GPULOG Across Different Hardware

To increase the performance portability of GPULOG, we translated it to a Heterogeneous-Compute Interface for Portability (HIP) [2] based engine (GPULOG-HIP hereafter) with an identical API to our vanilla CUDA variant. This interface enables seamlessly switching GPULOG implementations from NVIDIA to HIP kernels. The HIP version of GPULOG, while

Table 4. Context-Sensitive Program Analysis *(CSPA)* execution time comparison: GPULog (NVIDIA H100) vs. Soufflé (32 core AMD Milan CPU); input data from [4].

Dataset Name	Input Relation Size	Output Relation Size	Time (s) GPULog	Soufflé	Speedup
Httpd	Assign: 3.62e5 Dereference: 1.14e6	ValueFlow: 1.36e6 ValueAlias: 2.34e8 MemAlias:8.89e7	**1.33**	49.48	**37.2x**
Linux	Assign: 1.98e6 Dereference: 7.50e6	ValueFlow: 5.50e6 ValueAlias: 2.23e7 MemAlias:8.84e7	**0.39**	13.44	**34.5x**
PostgreSQL	Assign: 1.20e6 Dereference: 3.46e6	ValueFlow: 3.71e6 ValueAlias: 2.23e8 MemAlias:8.84e7	**1.27**	57.82	**44.9x**

Table 5. Comparison of GPULog and GPULog-HIP running times across various GPU vendors and models

Query	Dataset	GPULog (NVIDIA) H100	A100	GPULog (AMD) MI250	MI50
SG	fe_body	5.05	8.61	19.57	41.99
	BrightKite	3.42	6.79	14.00	30.05
	fe_sphere	2.36	4.64	8.48	19.426
CSPA	httpd	1.33	2.73	6.75	15.27
	linux	0.39	0.77	1.39	3.32
	postgres	1.27	2.68	6.79	14.55

Figure 6. Running time detail analysis of *CSPA* query using GPULog on various real world datasets on NVIDIA A100.

Table 6. Comparing Time Consuming Operation of GPULog on AMD EPYC 7543P 32-core Zen 3 CPU and NVIDIA A100 GPU. Each running time in table are in seconds.

# Tuples	Sort A100	Zen3	Merge A100	Zen3	Memory A100	Zen3
1,000,000	0.12	1.09	0.03	0.06	0.03	0.02
10,000,000	0.39	7.5	0.08	0.64	0.17	0.05
50,000,000	1.63	30.09	0.18	1.96	0.11	0.88
100,000,000	2.9	64.02	0.3	3.56	0.18	1.7
500,000,000	15.66	351.4	1.21	15.68	0.82	8.59

not matching the CUDA version's performance (due to missing libraries such as RMM [39]), offers broad compatibility with different GPU architectures. This enables rapid deployment on the exascale systems, for example, Frontier and Aurora [3, 38], which do not support CUDA.

To compare GPULog's performance across a range of data center GPUs, we ran *SG* and *CSPA* using GPULog on the NVIDIA H100 and A100 and the AMD MI250 and MI50. The outcomes of these tests are presented in Table 5. The NVIDIA H100, successor to the A100, features more SM units (114 vs. 108), double the FP32 cores per SM (128 vs. 64), and improved memory bandwidth (2TB/s vs. 1.5TB/s). Columns 4 and 5 of our results show GPULog benefits from these enhancements.

The last two columns illustrate GPULog's performance on AMD GPUs. The MI250, with 104 computational units (64 core per unit) akin to NVIDIA's SMs, shows comparable performance to the NVIDIA A100 in *CSPA* tests but lags in *SG* tests requiring more memory, performing at about half the A100's speed. The MI50, with half the capacity of the MI250, displays nearly half its performance, pointing towards the scalability of the approach. MI250 has similar total computational resources, it should have similar performance to the A100. However, the most important reason for the lower performance of the AMD MI250 is its dual chiplet design;
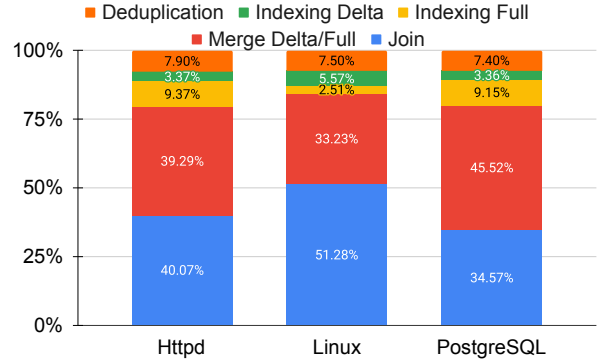
since GPULog is a single-GPU system, only half of the compute resources on the MI250 can be utilized, resulting in half the performance of the A100. Another reason is our use of the NVIDIA-specific RMM library in our implementation. Since AMD's ROCm does not support this library, we rely on manual memory pooling instead; we leave optimizing memory allocation in our HIP backend to future work.

Although fully implementing GPULog on a CPU via SIMD is possible, we believe such an implementation would not achieve similar performance due to the significant memory bandwidth differences between the two hardware architectures. Due to the extensive potential engineering work, we did not implement a full CPU version of GPULog. However, we did implement two GPULog's most computationally-expensive phases: sort, merge and implemented them using the latest version of Intel's oneTBB SIMD library (which uses CPU-specific vectorization) [19].

We ran both CPU and GPU implementations of sort and merge on different sizes of randomly generated tuples with two arities, collecting the total time of 100 runs. The results are presented in Table 6. Overall, in all test cases, the GPU version of both sort and merge functions is around 10x to 20x faster than the CPU version. We believe this is primarily due to the significantly better memory bandwidth offered by the GPU (1.5 TB/s vs. 0.19 TB/s). To investigate this, we

collected the buffer allocation and initialization times for these two operations and listed them in the last column, which shows the memory allocation and initialization of the buffer to hold tuples used in the experiment. For all input sizes, the A100 GPU is consistently 10x faster than the EPYC 7543. We note that the performance increases mirror the memory bandwidth differences between the CPU and GPU; indeed, because modern Datalog applications (such as *CSPA*) are memory-bound, we conclude GPUs offer an attractive implementation platform for modern Datalog engines.

7 Conclusion and Future Work

We presented GPULog, the first GPU-based Datalog engine able to achieve net-positive performance compared to SOTA Datalog implementations. Our library GPULog (implemented both for CUDA and HIP) enables modern Datalog implementation techniques, namely compilation to relational algebra, range-indexed iterated joins, and optimized n-way joins. GPULog is powered by the hash-indexed sorted array (HISA), a novel data structure we developed to support high-performance Datalog engines on the GPU. We then used HISA to build GPULog by employing two novel design points: temporarily materialized n-way joins and eager buffer management. We have evaluated GPULog extensively compared to both CPU and GPU-based state-of-the-art; our experimental results demonstrate improvements of up to $45\times$ compared to an optimally-tuned CPU-based engine on context-sensitive points-to analysis of PostgreSQL.

We notice recent work on running relational algebra and Datalog on multi-node systems, scaling them to thousands of nodes [14, 24, 25, 51] using fine-tuned MPI all-to-all communication [10, 11, 35]. Following this trend, one of our future projects is multi-node, multi-GPU programming coordinated via MPI. Such an effort necessitates a heterogeneous decomposition of work across nodes while minimizing communication. We plan to address this by leveraging the coarse-grained (task-level) parallelism inherent in specific applications. Additionally, we intend to extend GPULog to support monotonic aggregation and implement recent join algorithms, such as free join [54].

8 Acknowledgement

References

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

[2] AMD. Hip documentation, 2024.

[3] Argonne Leadership Computing Facility. Aurora supercomputer, 2024.

[4] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. Engineering a high-performance gpu b-tree. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 145–157, 2019.

[5] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 84–104. Springer, 2016.

[6] 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.

[7] 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.

[8] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. Dbsp: Automatic incremental view maintenance for rich query languages. *Proc. VLDB Endow.*, 16(7):1601–1614, mar 2023.

[9] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[10] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. Optimizing the bruck algorithm for non-uniform all-to-all communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 172–184, 2022.

[11] Ke Fan, Steve Petruzza, Thomas Gilray, and Sidharth Kumar. Configurable algorithms for all-to-all collectives. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12. Prometeus GmbH, 2024.

[12] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, Feb 2019.

[13] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092, 2020.

[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. Hashgraph—scalable hash tables using a sparse graph data structure. *ACM Trans. Parallel Comput.*, 8(2), jul 2021.

[16] Oded Green, Robert McColl, and David A Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340, 2012.

[17] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5(2):105–195, 2013.

[18] Jiaqi Gu, Yugo H Watanabe, William A Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In *Proceedings of the 2019 International Conference on Management of Data*, pages 467–484, 2019.

[19] Intel. oneAPI Threading Building Blocks (oneTBB). https://github.com/oneapi-src/oneTBB, 2024.

[20] JEDEC. High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/document_search?search_api_views_fulltext=jesd235, 2021.

[21] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 422–430. Springer, 2016.

[22] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. Brie: A specialized trie for concurrent datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 31–40, 2019.

[23] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 327–339, 2019.

[24] 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.

[25] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*, pages 288–308. Springer, 2020.

[26] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, 2005.

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

[28] 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.

[29] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. Dycuckoo: dynamic hash tables on gpus. In *2021 IEEE 37th international conference on data engineering (ICDE)*, pages 744–755. IEEE, 2021.

[30] David Maier, K Tuncay Tekle, Michael Kifer, and David S Warren. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. 2018.

[31] Carlos Alberto Martínez-Angeles, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. A datalog engine for gpus. In *International Conference on Applications of Declarative Programming and Knowledge Management*, pages 152–168. Springer, 2013.

[32] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[33] Raymond J Mooney. Inductive logic programming for natural language processing. In *International conference on inductive logic programming*, pages 1–22. Springer, 1996.

[34] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Naeris Netterville, Ke Fan, Sidharth Kumar, and Thomas Gilray. A visual guide to mpi all-to-all. In *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pages 20–27. IEEE, 2022.

[36] Nvida. Thrust: The C++ Parallel Algorithms Library . https://nvidia.github.io/cccl/thrust/, 2024.

[37] NVIDIA. CUDA C Programming Guide: SIMT. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture, 2024.

[38] Oak Ridge National Laboratory. Aurora supercomputer, 2024.

[39] RAPIDS. RMM: RAPIDS Memory Manager. https://github.com/rapidsai/rmm, 2024.

[40] RAPIDS Development Team. cuDF: Gpu dataframe library. https://github.com/rapidsai/cudf, 2021.

[41] Daniel Ritter and Till Westmann. Business network reconstruction using datalog. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, pages 148–152, Berlin, Heidelberg, 2012.

Springer Berlin Heidelberg.

[42] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. Bring your own data structures to datalog. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[43] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. ACM.

[44] 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.

[45] 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*, SIGMOD '16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery.

[46] Ahmedur Rahman Shovon. Public github repository of GPUJoin. https://github.com/harp-lab/usenixATC23, 2023.

[47] 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. IEEE, 2022.

[48] Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. Towards iterative relational algebra on the {GPU}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1009–1016, 2023.

[49] Evgeny Skvortsov, Yilin Xia, and Bertram Ludäscher. Logica: Declarative data science for mere mortals. In *EDBT*, pages 842–845, 2024.

[50] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation.

Proc. VLDB Endow., 12(2):141–153, oct 2018.

[51] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. Communication-avoiding recursive aggregation. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 197–208. IEEE, 2023.

[52] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[53] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Harry Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[54] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[55] 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*, pages 44–54, 2014.

[56] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. Optimizing parallel recursive datalog evaluation on multicore machines. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1433–1446, 2022.

[57] Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. Evaluating datalog over semirings: A grounding-based approach. *Proceedings of the ACM on Management of Data*, 2(2):1–26, 2024.