Column-Oriented Datalog on the GPU

Yihao Sun[1], **Sidharth Kumar**[2], **Thomas Gilray**[3], **Kristopher Micinski**[1]

[1]Syracuse University
[2]University of Illinois at Chicago
[3]Washington State University
{ysun67, kkmicins}@syr.edu, sidharth@uic.edu, thomas.gilray@wsu.edu

Abstract

Datalog is a logic programming language widely used in knowledge representation and reasoning (KRR), program analysis, and social media mining due to its expressiveness and high performance. Traditionally, Datalog engines use either row-oriented or column-oriented storage. Engines like VLog and Nemo favor column-oriented storage for efficiency on limited-resource machines, while row-oriented engines like Soufflé use advanced data structures with locking to perform better on multi-core CPUs. The advent of modern datacenter GPUs, such as the NVIDIA H100 with its ability to run over 16k threads simultaneously and high memory bandwidth, has reopened the debate on which storage layout is more effective. This paper presents the first column-oriented Datalog engines tailored to the strengths of modern GPUs. We present FVLOG, a CUDA-based Datalog runtime library with a column-oriented GPU datastructure that supports all necessary relational algebra operations. Our results demonstrate over 200× performance gains over SOTA CPU-based column-oriented Datalog engines and a 2.5× speedup over GPU Datalog engines in various workloads, including KRR.

Introduction

Datalog (Abiteboul, Hull, and Vianu 1995) has become a de-facto standard for reasoning across a breadth of fields, including deductive databases (Sáenz-Pérez, Caballero, and García-Ruiz 2011), program analysis (Bravenboer and Smaragdakis 2009), business analytics (Aref et al. 2015a), and knowledge representation (Nenov et al. 2015). For example, popular applications of RDF, such as OWL 2 RL ontologies (Motik et al. 2019) with SWRL rules (Horrocks et al. 2004), are easily transliterated into Datalog rules. A Datalog engine then executes these rules to a fixed point, materializing a database for subsequent querying. While early systems such as OWLIM (Kolovski, Wu, and Eadon 2010) and WebPIE (Urbani et al. 2010) were challenged by slow performance, modern Datalog engines enable scaling useful queries to internet-scale datasets (Ajileye and Motik 2022).

High-performance Datalog engines such as RDFox (Motik et al. 2014) and Soufflé (Jordan, Scholz, and Subotić 2016) are typically designed for CPU-based hardware, storing tuples in row-based format with minimally-locking data structures such as B-Trees and tries (Jordan et al. 2019a,b). However, row-based storage can face scalability challenges on modern CPUs, especially those with multiple Core Chiplet Dies (CCDs) that do not share caches. Accessing entire rows can lead to suboptimal cache alignment and increased memory access latency, particularly for tasks such as joins and indexing, which frequently access only a subset of columns. Additionally, these systems are not entirely lock-free, which can further limit scalability as core counts increase. Our benchmarks show that both engines' performance saturates at 32 cores and declines rapidly when scaling up to 64 cores.

Some CPU-based in-memory Datalog reasoners, such as VLog and Nemo (Urbani, Jacobs, and Krötzsch 2016; Ivliev et al. 2023), adopt a column-oriented tuple representation (Abadi, Madden, and Hachem 2008) along with compression techniques such as run-length encoding (Robinson and Cherry 1967). On CPU-based systems, the column-oriented approach trades space for time, enabling these engines to run on machines with limited RAM.

However, the performance claim between these two storage layouts shifts with the emergence of new hardware, particularly datacenter GPUs. These SIMD-like superchips, such as the NVIDIA H100, can execute more than 16 k threads simultaneously and are supported by 96 GB of high-bandwidth memory (HBM) (JEDEC 2021), and have significant potential to accelerate data-intensive workloads. Such massively parallel systems demand higher memory locality and fully lock-free code, making it challenging to migrate existing row-oriented Datalog engines to this hardware. In contrast, the column-oriented storage layout, with its smaller tuple size, naturally fits the SIMD architecture. This layout has already proven to be more cache-friendly and offers better memory locality for SIMD-enhanced CPU-based systems, as demonstrated in the database community (Ailamaki et al. 1999; Zukowski, Nes, and Boncz 2008), and is widely used in high-performance OLAP databases such as DuckDB (Raasveldt and Mühleisen 2019) and MonetDB/X100 (Boncz, Zukowski, and Nes 2005).

Due to the data-intensive nature of Datalog and the similarities between GPUs and SIMD processors, we believe that column-oriented storage could be a better fit for modern datacenter GPUs. In this paper, we present FVLOG, a column-oriented Datalog Engine backend designed for mod-

ern GPUs. Our contributions are as follows.

- We present the first-ever column-oriented relation-backing representation for GPU-based Datalog engines.
- We implement FVLOG, a CUDA-based GPU Datalog runtime library that supports efficient join, copy, deduplication, fixpoint computation, and other primitive operations for modern high-performance Datalog engines.
- We perform a thorough evaluation. Our results show over $200\times$ speedup compared to CPU-based column-oriented systems and $2.5\times$ faster performance than other GPU prototypes in both standard Datalog and knowledge graph reasoning workloads.

Preliminaries

Datalog and RDF Datalog restricts Prolog's to positive Horn clauses.It consists of a set of clauses in the form $H \leftarrow B_1, ..., B_n$, where the head H is derived if all body clauses $B_{1..n}$ are satisfied. A Datalog engine infers new facts (materializing an Intensional Database, IDB) from the rules and ground facts (also called the Extensional Database, EDB) provided as input. For example, transitive closure can be represented in following rules:

$$\begin{aligned} \textit{Reach}(x, y) &\leftarrow \textit{Edge}(x, y). \\ \textit{Reach}(x, z) &\leftarrow \textit{Edge}(x, y), \textit{Reach}(y, z). \end{aligned}$$

Datalog is chain-forward, recursively evaluating all rules to infer all possible facts until reaching a fixed point. One compelling application of Datalog is in materializing RDF knowledge graphs. RDF is a powerful framework for representing information about resources in the form of a directed, labeled graph. Edges in a knowledge graph may be represented via RDF triples using a predicate that relates two objects; for example, a knowledge graph for a family relationship may be represented as:

$$\begin{aligned} &< \texttt{Alice} \quad \textit{:parentOf} \quad \texttt{Bob} > \\ &< \texttt{Larry} \quad \textit{:parentOf} \quad \texttt{Alice} > \end{aligned}$$

One straightforward way to solve RDF reasoning problem in Datalog is by encoding the RDF triples as binary relation facts, then run recursive Datalog queries on it to materialize all information can be derived from the RDF graph. For example, above RDF triples can be encoded in Datalog as:

$$\begin{aligned} &\textit{parentOf}(\texttt{Alice}, \texttt{Bob}) \\ &\textit{parentOf}(\texttt{Larry}, \texttt{Alice}) \end{aligned}$$

Following transitive closure like Datalog rules can be used to materialize the ancestor relationship in the family graph:

$$\begin{aligned} \textit{ancestor}(x, y) &\leftarrow \textit{parentOf}(x, y). \\ \textit{ancestor}(x, z) &\leftarrow \textit{parentOf}(x, y), \textit{ancestor}(y, z). \end{aligned}$$

After above program reach the fixed point, the *ancestor* relation will contain all ancestor relationship in the family graph, we can directly query the *ancestor* to get the result.

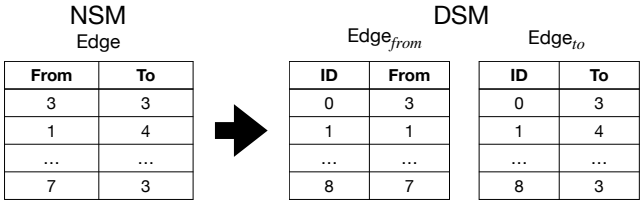

Figure 1: Converting *Edge* relation from NSM to DSM.

Decomposed Storage Model (DSM) Database records are traditionally stored as rows of n-ary tuples in a horizontal layout known as the N-ary Storage Model (NSM). Even today, most database management systems (DBMS) utilize NSM. However, some research demonstrated that storing database records via vertical columns could offer better performance (Weyl et al. 1975). This approach inspired the Decomposed Storage Model (DSM). Figure 1 shows an example of converting a NSM relation to DSM. An *Edge* relation is broken into two binary relations: $\textit{Edge}_{\textit{from}}$ and $\textit{Edge}_{to}$. Each of these binary relations includes an additional *ID* column that stores the original row number. For instance, the second row of the original relation, $(1, 4)$, is decomposed into $(1, 1)$ in $\textit{Edge}_{\textit{from}}$ and $(1, 4)$ in $\textit{Edge}_{to}$. This *ID* column, also known as the *surrogate column*, is essential for reconstructing the entire relation during join operations. Formally, a n-ary relation $R(x_0, ..., x_n)$ is decomposed to:

$$R_0(id, x_0), R_1(id, x_1), ..., R_n(id, x_n)$$

In the DSM model, accessing all values in the same column becomes straightforward—one simply needs to access each decomposed column relation. If the entire row is required, a join operation can be performed to combine the columns using the surrogate column. An n-ary DSM relation's row can be reconstructed as follows:

$$\Pi_{\neq id}(R_0 \bowtie_{id} R_1 \bowtie_{id} ... \bowtie_{id} R_n)$$

Transposing data into DSM facilitates vectorization, allowing multiple tuples to be processed in parallel, and improves cache efficiency with smaller tuple sizes (Ailamaki et al. 2001). These benefits are similar to those achieved by converting an Array of Structures (AoS) into a Structure of Arrays (SoA) in GPGPU programming (Pennycook et al. 2013). SoA is considered as the best practice for GPUs due to the enhanced memory coalescing and cache performance (NVIDIA 2024a). Given these advantages, recent surveys suggest that DSM could also enhance performance in GPU-based databases (Zeng et al. 2023), applying the same principles that benefit GPU workloads to DBMS.

While the DSM model offers excellent read performance, it often increases memory overhead due to the additional *id* column in each binary relation. Modern DSM-based databases, such as MonetDB/X100 (Boncz, Zukowski, and Nes 2005) and C-Store (Stonebraker et al. 2018), mitigate this overhead by storing tuples in lexicographic order and employing column compression techniques . However, the separation of each row's storage and the associated datastructure overhead make the write operation more expensive.

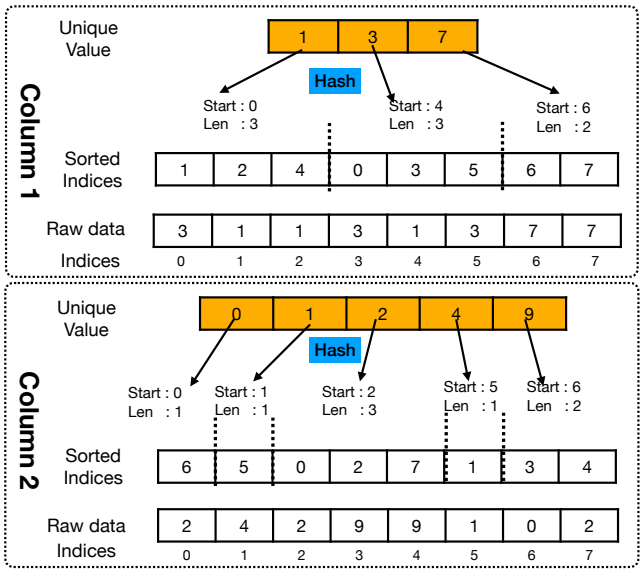

Figure 2: *Reach* stored in column-oriented layout on GPU.

This overhead is particularly problematic in Datalog, where the materialized IDB is often orders-of-magnitude larger than the input EDB.

Column-Oriented Relations on the GPU

In our approach, relations are stored using the DSM, where each relation is decomposed into columns, and each column shares an identical datastructure. Figure 2 illustrates our approach, particularized to the case of 2-ary relation. The column datastructure consists of three main components: a *raw data array*, *sorted indices*, and a *unique hashmap*. The bottom of each column in Figure 2 shows the raw data array, which stores the actual data of the relation in 32-bit integers ordered by logical tuple insertion time. A benefit of decomposing raw tuple rows into separate columns is that doing so facilitates aligned access on modern GPUs; GPU cores are typically 32-bit computation units organized into warps. To maximize parallel performance, all GPU threads in the same warp must execute the same instruction and access memory in a coalesced manner, with each thread accessing consecutive memory addresses (NVIDIA 2024b). When data is organized in a row-oriented manner, any relation with more than one attribute would exceed the size of a 32-bit integer, causing each GPU core to access non-consecutive memory addresses when processing data in the same column. These misaligned accesses significantly degrade throughput. By contrast, using column-oriented storage ensures that each column is stored in consecutive memory addresses, facilitating aligned access and aiding throughput.

The middle section of each column in Figure 2 represents the *sorted indices*, aligned with the surrogate column in the DSM model. Each element in this array is an offset into the raw data array, ordered by the values in that data. For instance, in column 2 of Figure 2, the 0^{th} value in the sorted indices is 6, pointing to the 6^{th} element in the raw data array, which is 0—the smallest value in this column. This sorted index acts as a database index, allowing quick location of corresponding value ranges during join operations, thus avoiding a full scan of the raw data array.

Pure binary search-based join operations on sorted indices can be inefficient on GPUs due to in-warp thread divergence caused by heavy conditional branching. To address this, we use a hybrid indexing approach that combines sorted indices with a *unique hashmap* at the top of each column in Figure 2. This design further enhances the efficiency of join operations. This hashmap stores run-length encoded values from the columns, with each unique value as a key. The hashmap value is a pair comprising the starting offset in the sorted indices and the count of occurrences. For example, in column 1 of Figure 2, the value 1 appears three times in the raw data array, starting at the 0^{th} element in the sorted indices, so the hashmap stores the pair (0, 3) for the key 1. This index structure enables fast hash joins and increases throughput.

We avoid using a pure hashmap for indexing because, unlike CPU, GPU hashmaps typically use linear probing and open addressing to resolve hash collisions. This approach, as seen in popular implementations such as cuCollection (cuCollection 2024), optimizes cache performance by ensuring continuous memory allocation. In Datalog, columns often contain many repeated values, leading to frequent hash collisions and degraded performance. Our benchmarks against GPUJoin (Shovon et al. 2023) demonstrate that our duplicated eliminated hybrid indexing approach is more efficient for real-world Datalog workloads.

Although our approach is inspired by the CPU-based column-oriented reasoner VLog, our design makes several important departures to optimize performance on modern GPUs, which we now describe.

Continuous Memory Layout To avoid re-evaluating known facts during fixpoint computation, Datalog implementations often adopt the semi-naı”ve evaluation. This method optimizes computation by managing each relation in three versions: *full* (all tuples), *delta* (tuples from the most recent iteration), and *new* (tuples generated in the current iteration), focusing on delta facts to streamline the process. However, this algorithm requires a write-intensive merge operation between *delta* and *full* in each iteration.

Due to the costly write operations associated with DSM relations, Datalog systems such as VLog avoid merging the *delta* generated in each iteration into the *full* relation. Instead, each datastructure object contains only the tuples generated within the same iteration. All relational algebra operations on the *full* relation are broken down into a series of operations on the *delta* in each iteration. While this design saves time during insertion, it introduces sequential overhead. When the number of iterations is low, this overhead is manageable on CPU-based systems. However, in real-world Datalog recursive queries, such as transitive closure on large graphs, hundreds of iterations are common. Excessive iterations cause the fragmented *full* relation to be scattered across memory, leading to poor memory locality and cache performance, which significantly increases overhead during joins.

The mitigating solution used by VLog involves *on-demand concatenation* to construct index during join opera-

tions. However, this approach remains unsuitable for GPUs. First, while on-demand concatenation helps reduce overhead during joins, it does not address the need to deduplicate *new* relation data with the fragmented *full* relation during semi-naïve evaluation. Second, this approach introduces memory management overhead due to the frequent, large memory allocations and deallocations required for temporary consolidated data structures in each iteration.

To address these challenges and better align with GPU architecture, we take the opposite approach. Instead of avoiding the overhead of data insertion, we accept it to maintain continuity in the raw data. This continuity ensures better data locality and cache performance, enabling more effective parallelism across the entire Datalog evaluation process. Moreover, due to the high memory bandwidth of GPUs, the parallel insertion of delta tuples into the *full* relation is not as expensive as on CPUs. Time-consuming operations such as sorting, scanning, and hash table construction can be efficiently parallelized (Satish et al. 2010; Green, McColl, and Bader 2012; Green 2021). Therefore, in FVLOG, we eagerly merge all delta tuples into the *full* relation to maintain the memory continuity and ensure better data locality.

Uncompressed Raw Data To save memory bandwidth, CPU-based column-oriented database systems use RLE compression on the raw data array. These systems are typically optimized for CPU in-memory processing. In contrast, we target datacenter GPUs, which offer large VRAM capacities of up to 192 GB, making raw column compression less beneficial. Thus, we compress only the indices and not the raw data. This approach avoids the overhead of decompressing raw data during parallel relational algebra operations such as joins and copies—wherein thousands of GPU threads need to iterate over raw tuples simultaneously—and maximizing parallelism and memory bandwidth of the GPU.

Schedule multiple rules per iteration VLog employs a one-rule-per-step variant of semi-naïve evaluation, where in each iteration, only one rule is applied. In this setup, if two Datalog rules contribute to the same relation in the same iteration–for example, in transitive closure, where both rules could contribute to the *Reach*–it results in two separate merge operations. In contrast, the original semi-naïve evaluation would have both rules contribute to the same relation simultaneously, requiring only a single merge. This is not a proper design for GPUs. In FVLOG, we adopt the original semi-naïve evaluation design.

Relational Algebra Operators

There are multiple ways to implement Datalog queries, such as using Binary Decision Diagrams (BDD) (Whaley and Lam 2004), SMT solvers (Hoder, Bjørner, and De Moura 2011), and Answer Set Programming (ASP) (Calimeri et al. 2017). However, state-of-the-art high-performance Datalog engines such as Soufflé (Aref et al. 2015b) achieve their high throughput via relational algebra kernels (Ceri, Gottlob, and Lavazza 1986). In this approach, a Datalog query is translated into a series of extended positive relational algebra ($\mathcal{RA}^+$) (Ullman 1983) operations, including join ($\bowtie$), selection (σ), projection (Π), and set union ($\cup$), along with an extended closure operator ($\mathcal{O}$), which computes the fixpoint. For example, to compute the transitive closure query, we can use the following $\mathcal{RA}^+$ expression:

$$\mathit{Reach} \quad = \quad \mathcal{O}(\Pi_{x,z}(\mathit{Edge} \bowtie_y \mathit{Reach}) \ \cup \ \mathit{Reach})$$

We now discuss the implementation of these $\mathcal{RA}^+$ operations in a column-oriented fashion optimized for GPUs.

Projection and Selection In FVLOG, the projection operator processes only surrogate columns instead of entire rows. The selection operator operates directly on the raw data array of the targeted column. Both RA operators can be implemented efficiently using NVIDIA's CCCL library.

Algorithm 1: Binary Join on hash indexed DSM relations

```
 1: Input: R_A, R_B are two DSM relations columns
 2: Output: the result of join stored in R_C
 3: ranges, matched_A ← allocate(size of R_A)
 4: for c in R_A, x is id of c parallel do
 5:    if R_B.hashmap[c] then
 6:       ranges[x] ← R_B.hashmap[c]
 7:       matched_A[x] ← x
 8:    else
 9:       ranges[x] ← (∅, ∅)
10:       matched_A[x] ← ∅
11:    end if
12: end for
13: filter out empty position in ranges and matched_A
14: total_size ← parallel sum sizes of all ranges
15: R_C ← parallel allocate(total_size)
16: pos_buf ← exclusive_scan(size of each ranges)
17: for n from 0 to R_C.size - 1 parallel do
18:    upper_bound ←
          j + 1 ≤ pos_buf.size ? pos_buf[j + 1] : total_size
19:    find j such that pos_buf[j] ≤ n < pos_buf[j + 1]
20:    A_id ← matched_A[j]
21:    B_id ← R_B.sorted_id[ranges[j].start +
             (n - pos_buf[j])]
22:    write (A_id, B_id) to R_C[n]
23: end for
```

Join Unlike the typical join operator in traditional row-oriented databases, we do not materialize the full join result within the join operator itself. Instead, we only return the matched surrogate columns of the join candidate relations. Join results are materialized only during a projection operation when the result columns are actually needed. This approach helps avoid unnecessary memory allocation for the full join result, saving both memory and computation time.

Algorithm 1 shows the implementation of join. The entire join can be divided into two main phases: computing join size (line 3 to line 16) and writing join results (line 17 to line 23). The first phase of the join process is illustrated in the left half of Figure 3. In this phase, each GPU thread parallelly iterates over the data column of the decomposed Reach_y (R_A in the algorithm). For each value, the thread queries the hashmap of Edge_y (R_B in the algorithm) to find

the matched surrogate values and the corresponding tuple ID ranges that share the same raw data value. Some values in $Reach_y$ may not have a match in $Edge_y$—for example, the bottom "7" in the Figure 3. In such cases, these unmatched values are filtered out by a filter function in line 16 of the algorithm. Next, by applying a parallel `reduce` function (line 14 of the algorithm) on the lengths of the matched ranges (indicated in red in the Figure 3), we compute the total number of join results, which in this case is 12. We then allocate memory for the results based on this computed size. Separating the join size computation phase from the result writing phase, rather than performing everything in a single loop (as is common in CPU-based engines), allows us to allocate memory for the join results in advance , which enables each thread to write to the join result without any lock contention.

There are two ways to collect the join result after computing the matched ranges. The first method involves parallel iteration over all matched ranges, where each thread writes a different number of tuples depending on the length of each range, which can cause data skew. For example, in Figure 3, the first matched range has a length of 3, while the second has only 1. The second method involves dividing the workload based on the output result, ensuring that each thread writes the same number of tuples, thereby avoiding data skew. However, this method requires extra searches within each thread to find the corresponding matched range, therefore most of CPU-based engines usually prefer the first method. However, most GPU algorithms favor the second style because reducing thread divergence improves performance when the thread count is large. In our implementation, we chose the second method for writing the join result.

We begin the second phase by performing a parallel exclusive prefix sum (line 16) on the lengths of each range identified in the previous phase to generate a result offset buffer, where each element represents the starting position in the result. These computed positions are marked in green in Figure 3. For example, nothing precedes the first range, so it has a result offset of "0". The third range has two preceding ranges with sizes of 3 and 1, so the resulting offset for the third range is 4. Then, we parallel iterate over all positions in the proclaimed result memory. Within each thread, a sequential search (which can be accelerated by binary search) identifies the corresponding matched position in the ranges, and the join result is written to the output buffer. For instance, thread 2 processes position number 2 in the result; by performing a binary search on the green buffer computed earlier, it determines that position 2 is between 0 and 3, matching the first range, which corresponds to the second to fifth values in the ID column of $Edge_y$. According to lines 19 and 20 in the algorithm, we compute that *Reach* ID 0 and *Edge* ID 2 should be written to the result buffer.

Union and Deduplication Implementing set union becomes more complex in DSM because the deduplication process during a join typically requires simultaneous access to entire rows. To effectively handle deduplication, we further extend $\mathcal{RA}^+$ with the difference ($-$) operator, which removes tuples from the left relation that have a matching tuple on the right. This extension allows us to efficiently

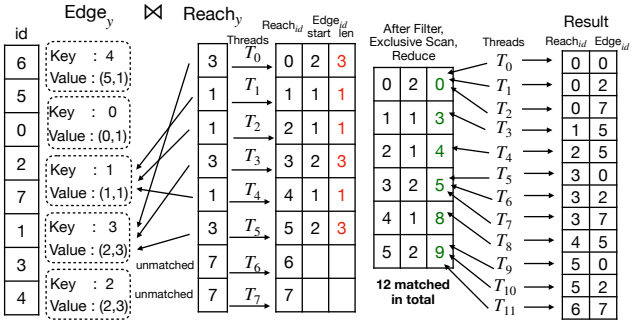

Figure 3: Example of $Reach \bowtie_y Edge$ paralleled on GPU.

manage deduplication. For example, the join operation in the transitive closure can be translated as:

$$\begin{aligned} New &= Edge_1 \bowtie_y Reach_0 \\ \Delta &= New - (New \bowtie_x Reach_0 \bowtie_{id,y} Reach_1) \\ Reach_{0,1} &= \mathcal{O}\left(\Pi_x(\Delta) \cup Reach_0, \Pi_y(\Delta) \cup Reach_1\right) \end{aligned}$$

Note that although we use the symbol $\cup$ here since the tuples have already been deduplicated, the set union operation is effectively a simple concatenation.

The join pattern revealed in this approach aligns with the classic *triangle join* problem, where three relations are joined to form a cycle. The complexity of evaluating such joins is tied to the AGM (Algebraic Graph Model) bound (Atserias, Grohe, and Marx 2013), which provides a worst-case estimate of the size of join results based on the sizes of the input relations. The AGM bound suggests that when data skew is present, any conventional binary join plan can become prohibitively expensive–a significant challenge for Datalog engines that rely on such plans. To illustrate this, consider the join within a deduplication process, which can be expressed in Horn Clause as follows:

$$Dedup(x, y) \leftarrow New(x, y), Reach_0(id, x), Reach_1(id, y)$$

Here, *New* represents the new tuples generated from the join between *Reach* and *Edge*. x, y, id form a join circle.

A classic solution is to use a trie-based join algorithm, such as Leapfrog Triejoin (LFTJ) (Veldhuizen 2014), widely used in CPU-based Datalog engines such as LogicBlox. However, LFTJ is unsuitable for GPUs because its Leapfrog search step is inherently sequential. Some other solutions such as generic join (Ngo, Ré, and Rudra 2014) and its column-oriented variant free join (Wang, Willsey, and Suciu 2023) have been proposed to address the triangle join problem in the context of traditional CPU-based databases. However, the recursive nature of free join makes it particularly challenging to implement on GPU.

Therefore, we take a different approach to handling triangle joins during deduplication. Instead of using tries, we tailor the process to our GPU-friendly datastructure. The details are presented in Algorithm 2. For simplicity, we demonstrate the deduplication process for a 2-arity relation, though the same approach can be extended to n-arity relations.

Algorithm 2 begins by computing the normal hash join between *New* and all the other columns in the full relation

Algorithm 2: Deduplication in FVLOG for a 2-arity relation

```
1: Input: New(x, y), S(id, x), T(id, y) are input relations
2: Output: Q is bitmap for matched New
3: for a in New_x parallel do
4:     if S.hashmap[a] then
5:         range_x ← S.hashmap[a]
6:     else
7:         range_x ← (∅,∅)
8:     end if
9: end for
10: match_id = range(R.size)
11: do the same for New_y and T generate range_y
12: remove i from match_id if either range is empty
13: for i in match_id parallel do
14:     if range_x[i] and range_y[i] are overlapped, Q[i]=true
15: end for
```

(the S and T in the algorithm). Before joining surrogate columns (lines 10-12), tuples with empty matched surrogate column ranges are marked for early elimination, reducing unnecessary computation. To prevent thread divergence and warp serialization, especially in the presence of data skew, the two join operations are separated into distinct parallel loops. After marking tuples (lines 13-15), another parallel loop efficiently checks for matches in the surrogate column.

While this approach may be less efficient in terms of memory usage compared with a solution like LFTJ, as it requires additional buffers equivalent in size to the newly generated relation, managing these buffers can be challenging and often requires techniques like best-fit allocation (Shore 1975). However, this lock-free design is particularly well-suited for GPUs, and the massive parallel speedup it offers makes the additional memory cost worthwhile.

Evaluation

We evaluate the performance of FVLOG through three key comparisons. First, we compare FVLOG with CPU-based column-oriented Datalog engines, VLog and Nemo, to demonstrate how GPU-optimized data structures can accelerate column-oriented Datalog. Next, we compare FVLOG with GPU-based row-oriented Datalog prototypes, GPUJoin and GDLOG, to highlight the superior performance of column-oriented storage on GPUs. Finally, we validate FVLOG by running the LUBM scenario in ChaseBench to demonstrate its applicability in knowledge graph reasoning.

Experimental Environment

All our experiments were conducted on a server equipped with an AMD EPYC 9534 and an NVIDIA H100. The AMD EPYC 9534 features 64 cores and 128 threads, supported by 500 GB of memory with a memory bandwidth of 0.43 TB/s. The NVIDIA H100 GPU includes 16,896 CUDA cores and 80 GB of HBM3 memory, offering up to 3.3 TB/s of memory bandwidth. The server runs Ubuntu 22.04 and GCC 11. For VLog, we used Rulewerk, a Java wrapper for VLog that provides additional language features. For Nemo, we utilized version 0.5.1. We used Soufflé version 2.4.1, with multithreading and compiler optimizations maximized. The RDFox version used was 7.1a, with all CPU threads enabled. All GPU tools were compiled with NVC++ in NVHPC 24.1.

Table 1: Running time (Second) of Same Generation Query. FVLOG and GDLOG are executed on NVIDIA H100, Nemo and soufflé are exectuetd on AMD EPYC 9534 (Genoa).

Dataset	Size	FVLOG	VLog	Nemo	Soufflé	RDFox
vsp_finan	552,020	7.52	4403	2172	151.5	257
fc_ocean	409,593	0.31	169.7	151.9	13.13	19.2
SF.cedge	223,001	1.80	1121	298.9	56.52	117
fe_body	163,734	1.85	173.9	555.4	48.18	126
CA-HepTH	51,971	0.55	313.7	147.7	20.12	20.1
fe_sphere	49,152	0.92	582.7	160.8	48.12	63.8

Column-Oriented Datalog Comparison

We first compared the performance of FVLOG with two CPU-based column-oriented Datalog engines, VLog and Nemo, using a simple yet representative Same Generation (SG) Datalog query. This query is a common pattern in Datalog reasoning and demands substantial computation time.

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

Results are presented in Table 1. The first column lists the names of the graphs used in this experiment, all of which come from the SparseSuite (Davis and Hu 2011) dataset. The size of each input data is reported in the second column. These graphs are real-world graphs extracted from diverse areas such as road systems, simulations, and SAT solving. This diversity ensures that our benchmark of the Datalog engines is unbiased. Columns three through five lists the running times for the benchmark candidates, while the last 2 columns include the state-of-the-art industrial Datalog engines, Soufflé and RDFox, as a reference.

The results demonstrate the significant performance gains achieved by running Datalog on GPUs. Under similar storage layouts (all candidates are column-oriented), FVLOG outperforms VLog and Nemo by a large margin. In all test cases, FVLOG on the H100 is at least more than 150 times faster than VLog and Nemo on AMD Genoa. Notably, on the *vsp_finan* dataset, which is a comparatively large input graph containing 552,020 edges, FVLOG is 584 times faster than VLog and 288 times faster than Nemo. Even when compared to Soufflé and RDFox, which are optimized multicore Datalog engines, FVLOG on the datacenter GPU still shows a significant performance advantage. On the *vsp_finan* dataset, FVLOG is 20 times faster than Soufflé, demonstrating the superior performance in large datasets.

Comparing FVLOG to SOTA Datalog Engines

To validate whether the column-oriented storage layout outperforms the row-oriented layout on GPUs, we compare FVLOG with two GPU-based row-oriented Datalog prototypes, GPUJoin (Shovon et al. 2023) and GDLOG (Sun et al. 2023). We use the transitive closure query mentioned in Section as the benchmark and also include Soufflé and RDFox

Table 2: Running time (Second) of Transitive Closure Query on FVLOG and GDLOG are executed on NVIDIA H100, soufflé are exectuetd on AMD EPYC 9534. 🐞 indicates a non-OOM crash observed running GPUJoin.

Dataset	FVLOG	GDLOG	GPUJoin	Soufflé	RDFox
vsp_finan	7.94	21.91	63.89	239.3	269
fe_ocean	10.07	23.36	🐞	292.2	507
usroads	9.55	17.53	57.89	243.1	268
com-dblp	3.35	14.30	🐞	233.0	569
Gnutella31	1.2	3.76	7.82	96.82	373
fe_sphere	0.53	0.93	1.16	25.02	25.1

as a reference. The running time results are shown in Table 2. Due to some bugs in the code, we were unable to obtain results for GPUJoin on the *fe_ocean* and *com-dblp* datasets. The results indicate that all GPU-based engines show significant performance improvements over the CPU-based engines. In sum, FVLOG is on average $2.5\times$ faster than GDLOG and $5.7\times$ faster than GPUJoin.

Our investigation showed that the comparatively low performance of GPUJoin is due to the use of a hash map for indexing used in this engine. Additionally, GPUJoin's need to compress entire rows into single 32-bit integers, restricting it to 2-arity relations, limits its versatility compared to FVLOG. FVLOG's advantage over GDLOG, despite both using hybrid indexing, lies in its column-oriented storage, which improves data locality and memory bandwidth utilization. The per-column processing in the DSM model further simplifies expensive operations like tuple sorting, enabling the use of parallel radix sort instead of merge sort, which is more efficient on GPUs.

Table 3: TGD reasoning time (seconds) comparison of different Datalog engine on LUBM. FVLOG(C) runs on AMD EPYC 9534 CPU, FVLOG(G) runs on H100 GPU.

Dataset	FVLOG(C)	FVLOG(G)	Nemo	VLog	RDFox
010	0.15	0.01	0.65	1.46	0.44
100	0.71	0.03	6.34	5.02	4.98
01K	6.47	0.16	62.32	165.8	56.8

Benchmark Knowledge Representation and Reasoning After benchmarking the basic Datalog reasoning queries, we further evaluated the performance of FVLOG on a KRR workload. We selected tuple-generating dependency (TGD) queries (excluding existential rules) on the LUBM dataset from the ChaseBench (Benedikt et al. 2017), a widely used dataset for evaluating Datalog-based KRR systems. The queries used in this test include both ST-TGD and T-TGD and were sourced from the example repository of Nemo (KBS 2024). The results are presented in Table 3. The first column lists the names of the sub-datasets used in this test, with the size of the input data increasing from top to bottom. The third column shows the running time of FVLOG on a H100 GPU, while the fourth and fifth columns display the running times of Nemo and VLog on an 64 cores EPYC 9543. In the last column, we also put the RDFox industrial row-oriented reasoner as a reference. By comparing the running times, we can conclude that combining the power of GPUs with performance-aware data structures enables FVLOG to significantly improve Datalog materialization times for KRR workloads. The maximum speedup achieved is close to $300\times$ in the largest input dataset compared to traditional CPU-based systems, Nemo and RDFox. This test, being more copy-intensive than previous join-heavy benchmarks, reveals that RDFox, even with 64 CPU cores, does not significantly outperform sequential reasoners. We attribute this to the memory-bound nature of the queries and the lack of data locality in row-oriented storage, which limits parallel performance.

To further investigate the contributions of performance-oriented data structures versus the benefits derived from the incredible memory bandwidth and core size of GPUs, we also developed a CPU variant of FVLOG. This variant employs similar data structures but leverages Intel's latest oneTBB (Intel 2024) to utilize the multicore resources on a datacenter CPU instead of GPU threading. The running times of the CPU version of FVLOG are listed in the second column of Table 3. The results show that the GPU version of FVLOG is at least $15\times$ faster than the CPU version. Considering that the memory bandwidth of the H100 is nearly $7.9\times$ times larger than the EPYC 9534, this indicates that the workload is memory-bound and that the performance gains on the GPU are primarily due to its high memory bandwidth. Additionally, when comparing the running times of the CPU version of FVLOG with Nemo and VLog, we observe around a $9.6\times$ speedup on the largest dataset. This suggests that performance-oriented data structures also contribute to about half of the overall improvement.

Conclusion and Future Work

In this paper, we demonstrate that a column-oriented storage layout is superior to a row-oriented layout on GPUs for Datalog processing. However, we also admitted that the uncompressed design of FVLOG and the persistence of surrogate columns result in higher memory usage. Despite this, the trend towards hardware with larger memory capacities and higher memory bandwidths makes FVLOG well-suited for future advancements. Looking ahead, another promising direction is to develop a cluster version of FVLOG to utilize the fast interconnects and advanced load balancing available in modern HPC environments. This would help overcome current memory size limitations and further enhance scalability and performance in distributed settings.

Acknowledgement

This work was funded in part by NSF PPoSS planning and large grants CCF-2316159 and CCF-2316157. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

References

Abadi, D. J.; Madden, S. R.; and Hachem, N. 2008. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 967–980.

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of databases*, volume 8. Addison-Wesley Reading.

Ailamaki, A.; DeWitt, D. J.; Hill, M. D.; and Skounakis, M. 2001. Weaving Relations for Cache Performance. In *VLDB*, volume 1, 169–180.

Ailamaki, A.; DeWitt, D. J.; Hill, M. D.; and Wood, D. A. 1999. DBMSs on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, 266–277.

Ajileye, T.; and Motik, B. 2022. Materialisation and data partitioning algorithms for distributed RDF systems. *Journal of Web Semantics*, 73: 100711.

Aref, M.; Kimelfeld, B.; Pasalic, E.; and Vasiloglou, N. 2015a. Extending datalog with analytics in LogicBlox. In *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management*.

Aref, M.; Ten Cate, B.; Green, T. J.; Kimelfeld, B.; Olteanu, D.; Pasalic, E.; Veldhuizen, T. L.; and Washburn, G. 2015b. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1371–1382.

Atserias, A.; Grohe, M.; and Marx, D. 2013. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4): 1737–1767.

Benedikt, M.; Konstantinidis, G.; Mecca, G.; Motik, B.; Papotti, P.; Santoro, D.; and Tsamoura, E. 2017. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 37–52.

Boncz, P. A.; Zukowski, M.; and Nes, N. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Cidr*, volume 5, 225–237.

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

Calimeri, F.; Fuscà, D.; Perri, S.; and Zangari, J. 2017. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1): 5–20.

Ceri, S.; Gottlob, G.; and Lavazza, L. 1986. Translation and optimization of logic queries: The algebraic approach. In *Proceedings of the 12th International Conference on Very Large Data Bases*, 395–402.

cuCollection. 2024. cuCollections (cuco), an open-source, header-only library of GPU-accelerated, concurrent data structures. https://github.com/NVIDIA/cuCollections.

Davis, T. A.; and Hu, Y. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1).

Green, O. 2021. HashGraph—Scalable hash tables using a sparse graph data structure. *ACM Transactions on Parallel Computing (TOPC)*, 8(2): 1–17.

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

Hoder, K.; Bjørner, N.; and De Moura, L. 2011. μZ–an efficient engine for fixed points with constraints. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, 457–462. Springer.

Horrocks, I.; Patel-Schneider, P. F.; Boley, H.; Tabet, S.; Grosof, B.; Dean, M.; et al. 2004. SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member submission*, 21(79): 1–31.

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

Ivliev, A.; Ellmauthaler, S.; Gerlach, L.; Marx, M.; Meißner, M.; Meusel, S.; and Krötzsch, M. 2023. Nemo: First Glimpse of a New Rule Engine. In Pontelli, E.; Costantini, S.; Dodaro, C.; Gaggl, S.; Calegari, R.; Garcez, A. D.; Fabiano, F.; Mileo, A.; Russo, A.; and Toni, F., eds., *Proceedings 39th International Conference on Logic Programming (ICLP 2023)*, volume 385 of *EPTCS*, 333–335.

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

Jordan, H.; Scholz, B.; and Subotić, P. 2016. 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*, 422–430. Springer.

Jordan, H.; Subotić, P.; Zhao, D.; and Scholz, B. 2019a. Brie: A specialized trie for concurrent datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, 31–40.

Jordan, H.; Subotić, P.; Zhao, D.; and Scholz, B. 2019b. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, 327–339.

KBS. 2024. Nemo Examples and Benchmarks. https://github.com/knowsys/nemo-examples/blob/main/chasebench/lubm/.

Kolovski, V.; Wu, Z.; and Eadon, G. 2010. Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In *International Semantic Web Conference*, 436–452. Springer.

Motik, B.; Nenov, Y.; Piro, R.; and Horrocks, I. 2019. Maintenance of datalog materialisations revisited. *Artificial Intelligence*, 269: 76–136.

Motik, B.; Nenov, Y.; Piro, R.; Horrocks, I.; and Olteanu, D. 2014. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28.

Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2015. RDFox: A highly-scalable RDF store. In *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14*, 3–20. Springer.

Ngo, H. Q.; Ré, C.; and Rudra, A. 2014. Skew strikes back: new developments in the theory of join algorithms. *Acm Sigmod Record*, 42(4): 5–16.

NVIDIA. 2024a. CUDA Best Practice Guide: Coalesced Access to Global Memory. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory.

NVIDIA. 2024b. CUDA Programming Guide: Programming Models. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

Pennycook, S. J.; Hammond, S. D.; Wright, S. A.; Herdman, J.; Miller, I.; and Jarvis, S. A. 2013. An investigation of the performance portability of OpenCL. *Journal of Parallel and Distributed Computing*, 73(11): 1439–1450.

Raasveldt, M.; and Mühleisen, H. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, 1981–1984.

Robinson, A. H.; and Cherry, C. 1967. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3): 356–364.

Sáenz-Pérez, F.; Caballero, R.; and García-Ruiz, Y. 2011. A deductive database with datalog and sql query languages. In *Programming Languages and Systems: 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings 9*, 66–73. Springer.

Satish, N.; Kim, C.; Chhugani, J.; Nguyen, A. D.; Lee, V. W.; Kim, D.; and Dubey, P. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 351–362.

Shore, J. E. 1975. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8): 433–440.

Shovon, A. R.; Gilray, T.; Micinski, K.; and Kumar, S. 2023. Towards iterative relational algebra on the {GPU}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 1009–1016.

Stonebraker, M.; Abadi, D. J.; Batkin, A.; Chen, X.; Cherniack, M.; Ferreira, M.; Lau, E.; Lin, A.; Madden, S.; O'Neil, E.; et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 491–518.

Sun, Y.; Shovon, A. R.; Gilray, T.; Micinski, K.; and Kumar, S. 2023. GDlog: A GPU-Accelerated Deductive Engine. *arXiv preprint arXiv:2311.02206*.

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

Urbani, J.; Jacobs, C.; and Krötzsch, M. 2016. Column-oriented datalog materialization for large knowledge graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Urbani, J.; Kotoulas, S.; Maassen, J.; Van Harmelen, F.; and Bal, H. 2010. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *The Semantic Web: Research and Applications: 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30–June 3, 2010, Proceedings, Part I 7*, 213–227. Springer.

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

Wang, Y. R.; Willsey, M.; and Suciu, D. 2023. Free join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data*, 1(2): 1–23.

Weyl, S.; Fries, J.; Wiederhold, G.; and Germano, F. 1975. A modular self-describing clinical databank system. *Computers and Biomedical Research*, 8(3): 279–293.

Whaley, J.; and Lam, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, 131–144.

Zeng, X.; Hui, Y.; Shen, J.; Pavlo, A.; McKinney, W.; and Zhang, H. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.*, 17(2): 148–161.

Zukowski, M.; Nes, N.; and Boncz, P. 2008. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, 47–54.