Accelerating Iterative Relational Algebra Operations with WebGPU

Jiaxin Lu, Sidharth Kumar
University of Illinois Chicago, Department of Computer Science
jlu73@uic.edu, sidharth@uic.edu

Introduction

Bottom-up logic programming (e.g., Datalog) runs via repeated relational-algebra kernels—selection, projection, join, and reorder—until no new facts emerge. While GPUs promise massive parallelism for these primitives, existing engines remain CPU-bound (8–16 threads) and struggle with deduplication and growing result sets.

We introduce a WebGPU-based hash-join pipeline that:

- Maps each RA primitive (hash-join, GPU radix sort, deduplication, set difference, insertion into the full relation) to its own WGSL compute shader
- Uses dynamic, growable GPU buffers and load-factor-tuned hash tables for memory management
- Drives a delta-only fixed-point loop, processing only newly discovered tuples each iteration

WebGPU Example: Matrix multiplication

WebGPU is an API specifically designed to expose modern GPU capabilities to web developers. It is designed to facilitate not only the rendering of graphics but also the execution of GPU computations within the browser environment.

Browser Computing Benefits:

- Instant accessibility; no installation
- Reduced server dependency; lower latency
- Enhanced data privacy and security
- Real-time interactivity and rich visualization

Matrix multiplication in WebGPU utilizes a compute shader to dispatch numerous parallel threads. Each thread computes the dot product of a row from the first matrix and a column from the second matrix. This process involves reading from input buffers and writing results to a GPU buffer, all in a single GPU pass, similar to the approach used in CUDA code.

WGSL Compute Shader Code

```
@compute @workgroup_size(16, 16)
fn main(@builtin(global_invocation_id) global_id:
vec3<u32>) {
  let row = global_id.y;
  let col = global_id.x;
  if((row < Width) && (col < Width)) {
    var Pvalue: u32 = 0;
    for(var i: u32 = 0u; i < Width; i++) {
      let m = M[row * Width + i];
      let n = N[i * Width + col];
      Pvalue = Pvalue + m * n;}
    P[row * Width + col] = Pvalue;}}
```

CUDA Code

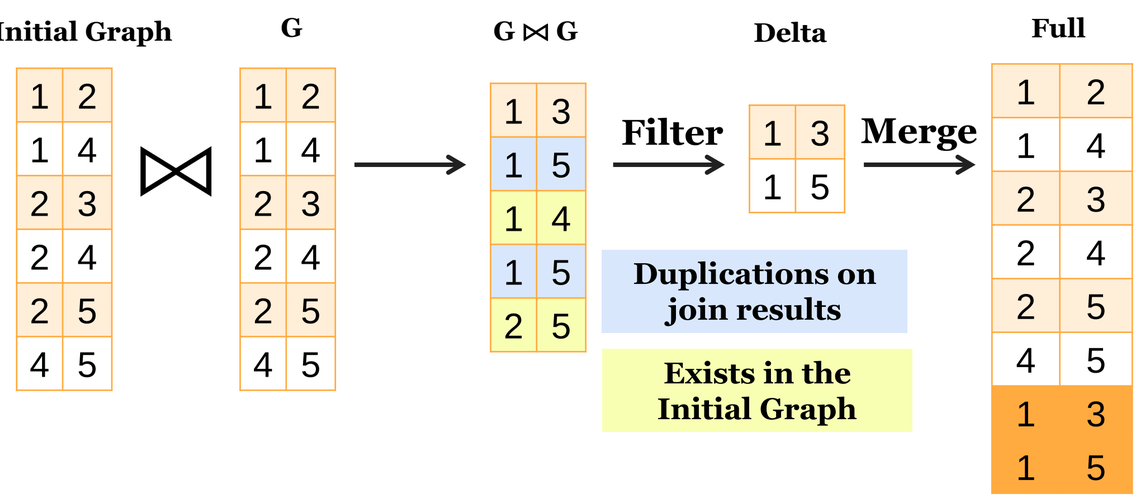
```
__global__ void matrixMul(int *a, int *b, int *c, int N){
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int temp_sum = 0;
  if(row < N && col < N){
    for(int i = 0; i < N; i++){
      temp_sum += a[row * N + i] * b[i * N + col];}
    c[row * N + col] = temp_sum;}}
```

Implementation

1. Array-Based Transitive Closure Computation Algorithm:
Starting from an initial graph, each round performs four stages in order:

- **Join** to compute new entries.
- **Count** to determine how many new entries are generated after joining.
- **Filter** to remove duplicates and retain only new entries.
- **Merge** to incorporate them into the full result relation.

Relation operations in a fixed-point setting (Iteration 1)



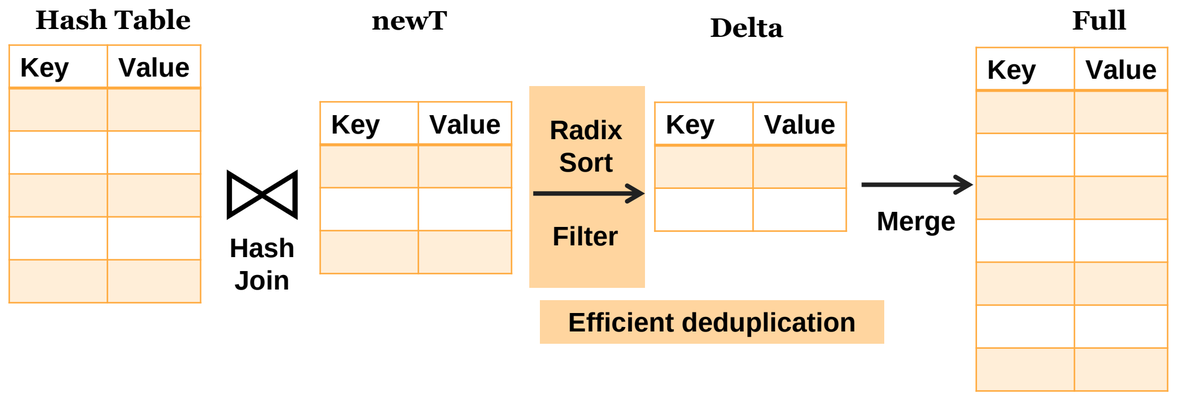
2. Hash Table-Based Transitive Closure Computation Algorithm:
WebGPU does not currently offer a built-in hash table data structure. Consequently, it is necessary to develop an appropriate hash table implementation within the WebGPU framework.

- **Build** a GPU-resident hash table in parallel. Populate the table and handle collisions (using Linear probing). Commencing with an initial graph, we utilize a hash table for the effective storage of the graph.
- Iterate through the processes of joining, counting, filtering, and merging until no additional entries are produced.

Implementation (Continue)

Our Problem:
To enhance the efficiency of the deduplication process, it is essential to conduct a sorting operation on the results derived from the join operation. By implementing this sorting technique, we can compare adjacent tuples rather than requiring each thread to examine every tuple in the join result individually. We have implemented a high-performance, four-way parallel radix sorting algorithm, which is expected to significantly reduce execution time compared to executing the program without this sorting approach. Unfortunately, following the implementation of the radix sort, the outcomes did not meet expectations, and we are currently working to identify the underlying issues.

Transitive closure computation (single iteration)



Experiments

We evaluated our method on small datasets and large datasets against the state-of-the-art CPU Datalog solver (Soufflé) and CUDA Hashjoin.

Array-Based WebGPU

Dataset	Type	Rows	TC size	Iterations	WebGPU (s)	GPUJoin (s)	Soufflé (s)
TG.cedge	U	23,874	481,121	58	10.8864	0.198	0.219
OL.cedge	U	7,035	146,120	64	5.4399	0.148	0.181
Small	U	10	18	3	0.1836	0.0342	0.007
Extra small	U	5	9	3	0.1893	0.0317	0.007

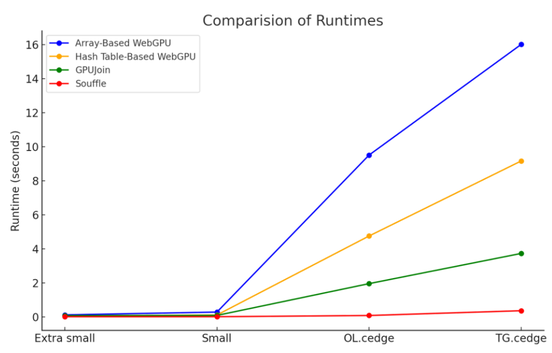
Hash Table-Based WebGPU without Sorting

Dataset	Type	Rows	TC size	Iterations	WebGPU (s)	GPUJoin (s)	Soufflé (s)
TG.cedge	U	23,874	481,121	58	7.7704	0.198	0.219
OL.cedge	U	7,035	146,120	64	3.0614	0.148	0.181
Small	U	10	18	3	0.0809	0.0342	0.007
Extra small	U	5	9	3	0.0565	0.0317	0.007

Hash Table-Based WebGPU with Radix Sort

Dataset	Type	Rows	TC size	Iterations	WebGPU (s)	Sorting Time (s)	GPUJoin (s)	Soufflé (s)
TG.cedge	U	23,874	481,121	58	99.63	69.1417	0.198	0.219
OL.cedge	U	7,035	146,120	64	40.644	30.764	0.148	0.181
Small	U	10	18	3	2.872	2.637	0.0342	0.007
Extra small	U	5	9	3	0.2170	0	0.0317	0.007

Conclusion



Runtime Performance Comparison of GPU-Accelerated and CPU-Based Relational Analytics

A hash-table-based WebGPU pipeline without sorting optimally balances performance and correctness, reducing array-scan time by 30% to 45% for large graphs and keeping small graphs' overhead under 0.1 seconds. However, adding a GPU radix sort incurs costs that exceed its benefits.

UNIVERSITY OF ILLINOIS CHICAGO