Aurochs: An Architecture for Dataflow Threads

Matthew Vilim
Stanford

Alexander Rucker
Stanford

Kunle Olukotun
Stanford

Abstract—Data analytics pipelines increasingly rely on databases to select, filter, and pre-process reams of data. These databases use data structures with irregular control flow like trees and hash tables which map poorly to existing database accelerators, leaving architects with a choice between CPUs—with stagnant performance—or accelerators that handle this complexity by relying on simpler but asymptotically sub-optimal algorithms.

To bridge this gap, we propose Aurochs: a reconfigurable dataflow accelerator (RDA) that matches a CPU asymptotically but outperforms it by over 100× on constant factors. We introduce a threading model for vector dataflow accelerators that extracts massive parallelism from irregular data structures using lightweight thread contexts. To implement this model, we add only a sparse scratchpad to an existing database accelerator—increasing area by 5%. We reformulate common data structures using dataflow threads and evaluate Aurochs on ridesharing queries—outperforming a GPU by 8×.

Index Terms—database, dataflow accelerator, CGRA, RDA, Plasticine, Gorgon, Aurochs

I. INTRODUCTION

Hardware accelerators are rapidly displacing CPUs for data-intensive workloads, but widespread deployment remains limited outside of all but the most ubiquitous domains (mainly machine learning). ML models are data-parallel algorithms that map well to throughput-based architectures like GPUs or TPUs. By contrast, the relational databases that feed these analytics pipelines rely on algorithms that are more difficult to accelerate due to irregular data and control flow. Efficiently selecting, filtering, and pre-processing data requires data structures like trees and hash tables that do not map well to existing throughput-centric architectures.

Databases form the backbone of streaming analytics applications [1], and analytics itself has moved beyond simple aggregate statistics to predictive ML models [2]. For example, sensor networks filter, aggregate, and join multiple data streams to correlate sensor readings that feed predictive ML models. These streaming analytics pipelines now process massive volumes of time-series and geospatial data (fig. 1); to meet real-time needs, asymptotically-optimal algorithms are essential.

Prior database accelerators like Gorgon [3] cope with these difficulties by resorting to simpler algorithms that are asymptotically sub-optimal but easier to accelerate. For example, sort-based joins and aggregations run in $O(n \log n)$ time while hash joins and aggregations scale linearly with table size. Software databases also use tree-based index structures to answer range queries in $O(\log n)$ time, while accelerators employ brute-force table scans. Thus, prior work exaggerates accelerators’ performance relative to CPUs, especially considering the small datasets imposed by cycle-accurate simulation. On large enough datasets, CPUs asymptotically outperform these accelerators because of algorithmic advantage.

Extracting parallelism from these algorithms is difficult, however. Traversing trees and hash tables requires pointer chasing with a loop-carried dependence that inhibits prefetching. Furthermore, hash tables have poor memory locality—due to sparse accesses when scattering records into buckets—and require collision resolution which occasionally has complex, data-dependent control flow.

Conventional vector architectures like GPUs limit the achievable vector and pipeline parallelism for applications with irregular control flow or sparse access patterns. Kernel launch enumerates all threads upfront [4] with no way to spawn, kill, or reorder threads at runtime. As a result, when pointer-chasing, GPUs cannot keep all lanes active by scheduling waiting threads to idle lanes when control flow diverges. Instead, GPUs serialize execution during branch divergence [5] even on the latest architectures [6], prohibiting high-performance irregular kernels. Moreover, GPUs’ inability to spawn new threads at runtime precludes following divergent paths through a tree in parallel. Finally, even if SIMD were a suitable programming model, GPUs waste energy on instruction fetch and decode and have large, power-hungry register files and caches [4].

Reconfigurable dataflow accelerators (RDAs) are a compelling alternative for data-intensive problems. RDAs outperform conventional architectures by unrolling applications spatially across a chip to extract massive vector, pipeline, and stream-level parallelism [7]. Programmer-managed scratchpads...
We implement Aurochs as a targeted, micro-architectural extension to Gorgon [3], an RDA for in-database machine learning. Aurochs adds memory reordering logic [8] to Gorgon’s SRAM scratchpads, increasing post-synthesis area by just 5%, and the design outperforms a GPU by 8× on a broad set of queries inspired by a ridesharing application.

This paper’s contributions are:

- A high-performance, low-overhead threading model for pointer-chasing problems on dataflow accelerators.
- A formulation of hash tables and trees as dataflow problems using this model.
- An end-to-end evaluation against a multi-core CPU and GPU on a diverse query set.

The remainder of this paper is structured as follows: §II provides background on Gorgon and relational databases. §III discusses Aurochs’ threading model and microarchitecture. §IV shows how we map high-performance hash tables and indices to this model. We evaluate Aurochs in §V and discuss related work in §VI.

II. Background

A. Relational databases

Relational databases represent structured data as tables of records—a record is a set of named fields (also called a tuple or a struct). Well-chosen database schemas maintain relations in normal form to eliminate redundancy. To denormalize tables, relational databases use joins to combine records from multiple tables. A join on two tables with a shared key concatenates records in both tables whose keys match. Joins are fundamental to relational data, and their use has grown beyond ad hoc analytic queries. Time-series databases now ingest streaming data in real-time and use sliding-window joins to correlate streams, and geospatial databases extend the notion of a join to answer spatial queries—a spatial join between two geographic keys yields points or regions between both tables that overlap.

As a result, asymptotically-optimal joins are critical to performance. Query planners choose the optimal join order and algorithm based on a query’s structure. Sort-merge joins pre-sort both tables on the join key in \(O(n \log n)\) time, followed by a linear-runtime merge operation. Hash joins build a hash table from one side of the relation and probe it with records from the other side. Each read and write to the hash table runs in constant time, so hash joins scale linearly with table size. Hash functions scramble keys to take skewed distributions to a uniform distribution, causing sparse, random memory access. So sort-merge joins outperform hash joins for small tables or if data is pre-sorted in indices due to better memory locality.

Databases also minimize query runtime by maintaining sorted indices on tables, which are often implemented with B-trees. For example, to answer a query for data within a time period, indices would extract records in \(O(\log n)\) time and avoid scanning the entire table. Geospatial databases generalize one-dimensional B-tree indices to two-dimensional structures like R-trees. For example, a spatial database can identify nearby points on a grid without testing every pair.
B. Gorgon

We implement Aurochs as a micro-architectural extension to Gorgon, a dataflow accelerator for in-database machine learning (DB+ML). Data-center operators avoid specialized accelerators to lower costs and simplify provisioning for the worst case [9]; Gorgon sidesteps this problem by co-locating DB and ML on a single fabric—a 20 × 20 grid of compute and scratchpad tiles, shown in fig. 2a.

All tiles are homogeneous and reconfigurable for use in either DB or ML kernels. Each compute tile is a 16-lane vector datapath that pipelines computations on streams across six stages that statically reconfigure to execute a single arithmetic or logical operation and provide cross-lane communication. Each memory tile is a reconfigurable scratchpad with 256 KiB of SRAM split across 16 banks. Gorgon lowers a SQL operator tree to a graph of these tiles connected by an on-chip interconnect with 5.1 TB/s bisection bandwidth [10]. On a DB+ML benchmark suite, Gorgon outperforms a multicore software baseline by 1500 x.

Records are central to Gorgon’s database processing. Each compute tile statically reconfigures a single datapath to filter, join, merge, aggregate, and partition record streams at line rate. To handle arbitrary record schemas and variable key lengths, Gorgon vectorizes record streams as shown on the right of fig. 2a—each record stays in one 32-bit lane with fields within the record processed sequentially across pipeline stages. This layout simplifies the microarchitecture by splitting and serializing keys wider than a lane to pipeline long comparisons between records across multiple cycles. As a GHz, each compute tile can process record streams at 64 GB/s; for even higher throughput, Gorgon parallelizes streams across multiple compute units to use HBM’s full bandwidth.

C. Capstan

Aurochs augments Gorgon’s SRAM scratchpads with a memory reordering pipeline proposed by Capstan [8], an RDA for sparse tensor algebra and graphs*. Scheduling hardware in the scratchpad executes requests out of order from issue queues to maximize throughput at the banks while maintaining memory consistency with rmw atomics. Figure 2b summarizes the memory reordering pipeline. Capstan frames sparse memory scheduling as a matching problem between 16 vector lanes and 16 SRAM banks and solves it using an allocator [11]. To resolve bank conflicts, requests in each lane’s issue queue bid to the allocator for bank access; the allocator attempts to find a maximal lane-bank pairing using combinational logic that fits in a single cycle. Following allocation, the issue queue immediately invalidates granted requests to avoid re-bidding from in-flight requests.

III. DESIGN

Aurochs accelerates pointer-chasing algorithms at the heart of data structures like trees and hash tables. To motivate Aurochs’ design, we discuss the characteristics of these workloads, why existing vector architectures underperform on these problems, and how Aurochs is better. Most importantly, our proposed design is lightweight and piggybacks on vector record-processing hardware native to database accelerators. This simplicity allows other database accelerators to adopt our design with very few micro-architectural changes.

A. Dataflow threads

To understand Aurochs’ threading model, it is important to understand why GPUs perform poorly when pointer-chasing. Parallel pointer-chasing Pointer-chasing along a linked list or tree is naturally serial due to the loop-carried dependence that prevents prefetching the next node. To overcome this inherent limitation, parallel implementations must either fetch enough data from each node to mask memory latency—like B-trees—or traverse many lists in parallel. The former approach is not always viable, and the latter does not map well to vector architectures like GPUs.

Consider a parallel hash table probe, shown in fig. 3. The goal is to traverse an array of linked lists and check each list for a different search key—hash tables do this when probing for multiple keys in parallel. On a GPU, a shared scratchpad stores the lists’ nodes q, and threads within a warp navigate independent lists with per-thread data stored in a large register file @. When fetching the next node from shared memory, sparse random reads cause bank conflicts @ when threads stall waiting on memory, GPUs rapidly context switch by scheduling a new warp. This behavior is especially prominent when walking lists because each thread runs few instructions before stalling when loading the next node. GPUs enumerate all warps in a thread block upfront by reserving space in register files to hold per-thread local state [4]. So, this register file balloons when storing enough warps to keep the GPU active [4,12].

Unfortunately, this approach couples each thread to one lane in a specific warp, which lowers throughput during inter- and intra-warp control flow divergence [13]. When threads within a warp diverge while comparing keys or checking for the list’s end, GPUs serialize both sides’ execution by predicated execution with lane masks [13]. Nvidia introduced independent thread scheduling [6] in Volta to allow divergent threads within a warp to interface execution at a finer granularity; however, execution remains serial, and threads remain locked to a warp. When threads across warps diverge, barrier synchronization prevents early finishers from continuing until all warps complete the search and reconverge.

*Both Gorgon and Capstan are derivatives of the Plasticine [7] RDA.
Consequently, straggler threads in a block can keep many warps scheduled with only a few lanes active [13]. We profile [14] a CUDA hash join implementation [15] on a V100 GPU and show a warp execution efficiency [16] of 62% during the build phase and 46% during the probe phase, indicating most lanes are idle and the GPU is not memory-bound.

Reserving storage for thread state upfront prevents spawning or killing threads at runtime—features that are useful when traversing data structures. Pointer-chasing threads have data-dependent lifetimes and require per-lane control flow to avoid serializing execution during branch divergence. For example, when a thread finishes early by finding the search key or reaching a list end, hardware should kill it and rapidly schedule another to its lane. In addition, forking threads to spawn children enables pointer-chasing multiple paths through a tree simultaneously.

**Thread records** To meet these demands, we propose a new threading model. In contrast to GPUs, dataflow architectures unroll computations spatially across distributed compute and memory tiles. GPUs’ register files are both area- and power-hungry [4]—but above all, they are unnecessary for pointer-chasing applications. Navigating irregular data structures requires very modest amounts of per-thread state—often, just several 32-bit words.

Consequently, we encapsulate per-thread state in small, ephemeral records that stream through pipelines—a natural fit for a database accelerator. Each record is a sequence of 32-bit fields that fully captures per-thread local state; pipeline stages mutate thread state as records flow through Aurochs’ compute and scratchpad tiles. Database accelerators already natively support computations over record streams, so Aurochs repurposes this vector record-processing hardware to pointer-chase in parallel.

Moving thread state out of register files and into the dataflow decouples threads from a lane or warp, enabling hardware to spawn new threads at runtime or refill idle lanes when threads terminate or branches diverge. In our design, control divergence on a vector dataflow accelerator simply filters threads on a predicate to split divergent threads into two parallel pipelines that execute both sides of the branch concurrently. Figure 4 shows how we transform control flow to dataflow by filtering thread contexts on a branch predicate. This approach handles data-dependent control flow or early termination without stalling the primary pipeline and quickly refills idle lanes with upstream threads. Moreover, Aurochs’ ability to spawn threads dynamically enables fork-join parallelism while traversing trees—a thread can fork itself to walk multiple paths down the tree.

To keep pipelines full, threads execute out of order because threads in separate pipelines can only synchronize or communicate periodically using shared scratchpads. Therefore, we further restrict our programming model to allow cross-thread communication only through atomic read-modify-write (rmw) scratchpad access. This choice decouples threads’ execution order entirely, letting hardware aggressively reorder threads in the dataflow to maximize throughput by minimizing bank conflicts. By contrast, GPU scratchpads use arbitrated crossbars and only schedule accesses within a single vector to resolve bank conflicts [5].

Due to the restrictive nature of this programming model, high-performance algorithms require careful mapping to maximize locality using on-chip memories but minimize cross-thread communication to avoid bottlenecks on scratchpad synchronization. So, in this paper, we do not expose this programming model through a user-facing, high-level language. Instead, we map the database kernels ourselves and expose them to users as SQL operators with parallelization knobs.

**Threading primitives and microarchitecture** Loose coupling and restricted communication between threads are important for good compute utilization and scalability. GPUs store thread state in large memories which causes communication hotspots and limits throughput. Because our thread contexts are ephemeral, their backing storage is not fixed to a single memory or register file, so smaller, distributed memories provide higher aggregate bandwidth. By design, Rdas tolerate long latencies by pipelining computations without loop-carried dependencies. However, applications with a loop-carried dependence (e.g., linked list traversal) instead require thread-level parallelism—keeping thousands of independent threads in-flight. Full hardware utilization is possible even with arbitrary on-chip latencies as long as there are enough threads to fill the pipeline. Consequently, our architecture easily scales to large grid sizes despite increasing communication delays between tiles.

Aurochs outperforms more conventional architectures when pointer-chasing by keeping pipelines full—even though pointer-chasing threads have variable, data-dependent lifetimes. A common pattern in these applications is a variable-runtime while loop to walk paths through a data structure. For illustration, fig. 5a shows how a linked list traversal maps to our dataflow using a cyclic datapath. Compute and scratchpad tiles are loosely timed through a streaming ready–valid interface with skid buffering. Each list node has a key and next node pointer—an index into a scratchpad that stores...
Dataflow overview and microarchitecture

The list nodes themselves. A compute tile executes the branch by filtering a stream of thread contexts; nodes in the middle of a list update their thread state by fetching the next node from the scratchpad. This process repeats until the list end is reached. When a thread terminates, empty buffer space ripples to the head of the pipeline, and a new thread enters to replace it.

This model’s simplicity requires only a few basic primitives with minimal hardware cost that already exist in database accelerators like Gorgon (fig. 5b). A filter is a native sql operator and splits a record stream in two using a predicate function. Filters implement branches but can also terminate threads by dropping one stream. A merge combines records from two streams into a single stream to recombine threads after branch divergence; typically one of the streams takes priority to avoid deadlock if there is cyclic dataflow like in fig. 5a. A map mutates thread state by adding and dropping fields or applying a function to each field; a common map operation is an atomic rmw scratchpad operation used to synchronize and communicate between threads. A fork spawns a batch of threads from a single thread; this ability is especially useful when a thread must walk multiple paths through a data structure.

Thread compaction reschedules threads to empty lanes when control diverges to keep hardware active. In our programming model, the order of threads within a stream does not imply implicit state (i.e., a thread index) within the dataflow. Consequently, any scheduling of threads to lanes is permissible—even reordering within a stream. Our abstract machine computes over record streams which can be implemented as either a vector or scalar datapath. Aurochs inherits Gorgon’s vector record hardware which lays each record on one lane and computes across fields in-time (fig. 2). Record schemas are statically reconfigurable, but all records in a stream share the same schema.

Figure 4 shows the high-level datapath within the compute tile. Two record buffers at the head of the pipeline feed the selector which chooses records from both streams and passes them through a shuffle network that interchanges records on adjacent lanes. Figure 5c illustrates compacting three sets of filtered records. The compute tile permutes lanes using the shuffle network to collapse empty lanes until all records are adjacent. A barrel shifter then rotates vectors using an accumulator to track how many lanes to shift. Finally, a downstream compute tile writes these compacted vectors into its record buffers in a dense format.

Record stream lengths are data-dependent and unknown until runtime; however, record streams are self-timed at each tile and do not require global control signals. When a tile receives a stream end signal from all its upstream producers, it generates a single pulse to indicate stream end to its downstream consumers. To avoid deadlock, when there are cycles in the compute graph like fig. 5a, the tile first signals stream end on the cyclic path and waits for the token at its input to ensure the cyclic pipeline has emptied before signalling stream end to consumers on the non-cyclic path.

Dataflow mapping To illustrate this threading model’s flexibility, we discuss mapping three irregular problems to our dataflow (fig. 6): hash table probe, B-tree search, and hash table build. Each algorithm’s dataflow mapping is drawn alongside annotated pseudocode.

For clarity, we write the pseudocode as a scalar while loop; however, loop iterations are fully independent, and Aurochs’ hardware provides no iteration ordering guarantees because threads are free to bypass one another. When unrolled across Aurochs’ vector lanes, 16 loop iterations run in parallel and simd lockstep within a compute or scratchpad tile. To break lockstep execution during control flow divergence, compute units test the branch condition and filter threads on the outcome into two parallel streams that execute both sides of the branch asynchronously. After control divergence, both pipelines compact their threads, making space for upstream threads to fill empty lanes.

Naturally, we configure each pipeline with only the necessary hardware to execute its side of the branch. Ofentimes, the required thread state (i.e., record fields) is branch-dependent, so Aurochs adds, drops, mutates, or permutes records as they move between pipelines based on data-dependent control flow. Micro-architectural support for record filtering and lane compaction are discussed above.

Revisiting hash table probe (fig. 6a), each thread fully captures its iteration state with a three-field record: the search key, current node, and a pointer to the next node. A banked sram scratchpad holds the array of list nodes. Compute
tiles compare the search key against each thread’s current node key; threads with matching keys exit the loop through an output stream, and mismatching threads update their current node with a sparse gather from the scratchpad using the next node pointer. Threads recirculate through this pipeline until their search key is found or the list end is reached. When a thread terminates, a compute tile filters it and immediately refills its lane with an upstream thread. Ignoring bank conflicts, this pipeline runs at line rate and tolerates long on-chip network delays despite the loop-carried dependence because we pipeline and overlap many independent searches at once.

B-tree search (fig. 6b) is similar to hash table probe but forks threads while traversing a tree. B-trees are balanced search trees that store internal nodes in blocks to improve memory locality. On Aurochs, the per-thread state is the search key and current node which holds a pointer to an array of child nodes and their key range. If the search key falls within the range, each thread loads its child tree nodes from dram and inserts them into the pipeline as new threads with the same search key; this process repeats until reaching the leaf nodes. The block size hides dram latency, keeping the pipeline full and allowing Aurochs to simultaneously walk multiple search paths through a B-tree at line rate.

**Hash table build (fig. 6c)** is an example of cross-thread communication using atomic scratchpad access. In this problem, a hash table uses chaining to resolve collisions by prepending a stream of keys into an array of collision lists. A lock-free, linked list prepend is a simple compare-and-swap (cas) that exchanges the old head and new head pointers if no other thread has prepended to the list since last read [17]. When mapping to Aurochs, the linked list head pointers and node data are physically split across two independent scratchpads because their access patterns differ. We reconfigure the list pointer scratchpad to implement an atomic gather-scatter cas. If cas fails (indicating a stale value), threads recirculate for retry with the latest head pointer. After each failed attempt, threads must scatter the current head pointer into the new node before retrying cas to ensure the list is always in a consistent state. Ignoring collisions, preempts run at line rate.

**B. Sparse reordering pipeline**

To reduce bank conflicts, we extend Gorgon’s scratchpad with Capstan’s sparse memory scheduling pipeline [8] discussed in §II∗ (fig. 2b). Each thread encodes its request’s data and address fields in a single record, and the scratchpad buffers thread vectors in issue queues at its head. An allocator reads all requests in parallel, matches lanes to banks, and grants at most one request per lane. Capstan’s issue queue design is low-overhead and stores only the request bank in registers for parallel readout by the allocator—with the remainder stored in a register file.

Our implementation here differs slightly from Capstan’s, however. Capstan maintains stream ordering by scheduling accesses across vectors out of order but dequeuing vectors in order. Consequently, it is susceptible to head-of-line blocking caused by straggler requests at the head of the pipeline. Capstan copes with this behavior by deepening the issue queues and employing multiple rounds of priority-based allocation that favors older requests. Critically, by decoupling thread execution order and encapsulating thread state in records, our programming model allows reordering threads in the dataflow. All requests execute atomically, so any ordering is permissible. Consequently, our issue queues are half as deep as Capstan’s because they invalidate granted requests, immediately freeing the lane’s slot for a new thread. As a result, with 16 lanes and a scheduling depth of eight, in each cycle the allocator considers up to 128 requests for execution.

Scratchpad banks are dual-ported srams, so we schedule reads and writes independently from split issue queues and allocators. Each scratchpad services two streams—which can each be configured as a read, write, or read-modify-write stream. When configured for rmw streams, small reconfiguration logic fuses both streams’ allocators and crossbars to schedule both streams from a single widened pipeline. Read-modify-write atomics like cas use a two stage pipeline with

∗Aurochs does not include Capstan’s sparse iteration primitives.
a forwarding path from write to read to enable back-to-back rmw operations to the same offset in a bank.

IV. Data structures

Aurochs’ hash table and tree implementations play to dataflow accelerators’ strength—throughput—but sidestep the complexity and irregularity of many cpu implementations. Arbitrary data structures require fine- and coarse-grained dynamic memory allocation and must remain consistent even during concurrent reads and writes. cpu implementations are control-heavy and rely on complicated locking schemes for parallelism and concurrency. Instead, we favor persistent data structures that are append only. This approach simplifies memory management by avoiding fine-grained deallocations and concurrency by enabling lock-free prepends using cas. We implement these structures fully with reconfigurable hardware without software fallbacks.

A. Hash table

High-performance hash tables are the basis of hash joins and hash-based aggregations. When joining, software databases partition tables on their join keys for parallelism and locality [18]. This pre-processing step guarantees partitions hold disjoint keys and enables joining partition pairs in parallel without synchronization. cpus size these partitions to be small enough to fit in the last-level cache (llc) to reduce sparse dram access. To join partitions, databases build a hash table from one partition and probe the table with records from the other partition for matches.

We implement the same two-phase approach in Aurochs—partitioning tables to dram first and building hash tables on each partition in scratchpads. For stream-level parallelism, a network of compute tiles partitions record streams on the low-radix bits of their hash key to feed multiple hashing pipelines in parallel. Radix partitioning on the hash load-balances parallel hashing pipelines regardless of skew because hash functions naturally generate uniform distributions. After partitioning, each stream iterates over partitions building hash tables on-chip without cross-stream communication.

Hash table on-chip Our hash table implementation attempts to keep keys entirely on-chip in scratchpads but provides fallback if a partition exceeds scratchpad capacity. During the hash partitioning phase, we choose the partition count so that the expected partition size matches scratchpads’ size (256 KiB). However, the resulting partition sizes are probabilistic, and outliers can far exceed scratchpad capacity. Figure 7a shows how on-chip hash tables map to Aurochs dataflow. The hash table is an array of linked lists that tracks buckets’ collisions. One scratchpad stores buckets’ head node pointers @—offsets into another scratchpad that holds the list nodes themselves @.

On ingress, the pipeline stamps a field in each thread’s context with an incrementing counter that indicates the thread’s reserved slot in the node scratchpad. When this value falls within the scratchpad’s address space, it denotes a pointer into the scratchpad; otherwise, it implicitly refers to an offset into a pre-allocated overflow buffer in dram @. Threads with a count larger than scratchpad capacity flow through an alternate path and are written to the overflow buffer; a base–offset calculation converts between either address space as threads move through the pipeline. Threads from both paths rejoin by merging to a single stream again before accessing the head node scratchpad; after writing to their reserved node in sr am or dram, threads attempt to link themselves into a collision chain using cas as described in §III-A. This straightforward control mechanism enables walking linked lists that are backed by both sr am and dram, transparently fetching nodes from either memory.

To read entries out of the hash table, a parallel read pipeline (not drawn) walks these lists searching for keys as described previously. Aurochs’ lock-free implementation guarantees buckets always appear consistent to all threads, allowing simultaneous reads and writes at line rate. This ability is critical for low-latency stream joins [19] where two streams build hash tables with the other’s records that they simultaneously probe with their own. Since Aurochs’ memory reordering pipeline independently schedules reads and writes to dual-ported sr am banks, concurrent hash table access has no performance impact and provides both streams with an optimal schedule.
Log-structured merge-tree and immutable B-tree data structures and the number support key deletions by adding implement. These threads to improve do not could.

**Hash partition off-chip** Figure 7b summarizes our approach to mapping hash partitioning to Aurochs. Sparse DRAM writes are unavoidable when recording scans into partitions with a hash function. Instead, our hash partition implementation avoids sparse reads when building hash tables from each partition by storing partitions in a dense format. Each partition is a linked list that stores an array of records per node @ to improve locality and mask memory latency when loading partitions from DRAM. Partitions are too large to hold on-chip, so we store the list nodes themselves in DRAM. And partition size is data-dependent, so Aurochs allocates and prepends list blocks at runtime. On-chip scratchpads store partition metadata which includes head pointers to list nodes in DRAM and the number of records in each partition’s head block @.

Our dataflow mapping handles exceptional control flow and dynamic memory allocation without stalling the partitioning pipeline. Compute units hash records within a stream and append the hash as a 32-bit field that flows with the thread through the pipeline. Threads index a scratchpad with their hash to perform a sparse, atomic fetch-and-add (FAA) to each partition’s count that tracks the number of records in a partition’s head block. A count less than the block size indicates the thread’s head block has free space, so threads scatter their keys to this slot in DRAM. A count equal to the block size indicates a thread is the first to encounter a full block @. These threads flow through a parallel pipeline that allocates and prepends a new block to the partition using CAS as described in the previous section. A count greater than the block size indicate a thread’s head block is full but another thread has already begun allocation. These threads occur rarely and recirculate continuously until the allocating thread completes by resetting the count to zero. Meanwhile, other threads with available partition space bypass stalled threads.

**B-tree**

To avoid table scans when range-querying data, database indices maintain an ordered map of keys pointing to records in an unordered table. Indices are especially important for time-series data because analysts typically query for relevant data over a narrow time period. Relational databases often implement indices with B-trees that keep sorted leaves with pointers to the full record in the indexed table (fig. 8).

### Index insertions

Databases that ingest streaming data rebuild indices continuously, keeping recent data in memory for real-time analytics. Keeping trees’ characteristic $O(\log n)$ asymptotic performance requires rebalancing during insertions to keep all leaves at the same height. Indices are read-optimized structures—their performance degrades as writes increase due to the cost of index updates. For write-heavy workloads, databases batch index updates to amortize their cost. And to enable concurrent readers and writers during insertions, many software databases implement elaborate tree locking algorithms or use lock-free data structures like skip lists [20].

We sidestep this complexity by implementing *log-structured merge-trees* (lsm). Lsm trees are append-only data structures that maintain a list of exponentially-growing immutable trees [21]. To query the entire structure, lsm trees search all internal trees. They batch inserts by bulk loading small record batches into a new tree and recursively merging the list of trees to maintain the exponential size difference. Batch size is a trade off between index update latency and work amortization. Lsm trees require only merge sort to implement—a kernel that Aurochs inherits from Gorgon. This simplicity makes them highly amenable to hardware acceleration. Gorgon’s sort implementation already includes partitioning for stream-level parallelism, tiling in scratchpads to sort without off-chip traffic, and high-radix merges to conserve DRAM bandwidth.

Each tree inside the lsm is an immutable B-tree that we bulk load into a single array by first sorting the leaves in $O(n \log n)$ time and then building the internal nodes in linear time (fig. 8). When the lsm tree’s first internal tree grows to the same size as the second tree, we merge both trees’ leaves in a new buffer with linear time and rebuild the internal nodes from scratch. A single lock-free update to the head list pointer replaces both trees with the newly merged tree.

lsm trees have many desirable properties. Since the internal trees are immutable, they provide natural concurrency to readers and writers without locking mechanisms; read queries traverse internal trees and background index updates write to new buffers while merging trees. Bulk-loading trees during insertions amortizes index updates and provides high-write throughput. An additional benefit of lsm trees for time-series queries is that the tree list acts as a secondary index on time, limiting the number of internal trees to search.

It is important to note that table and index insertions are not the same as sql transactions. Transactions are a sequence of related record insertions, updates, or deletions that must perform atomically. Aurochs only guarantees eventual consistency when ingesting data. Queries will always observe consistent
Fig. 9

R-trees are multi-dimensional index structures to map geographic data. For example, ridesharing applications use spatial indices to match nearby riders and drivers without testing all possible pairs. Spatial indices like R-trees generalize one-dimensional index structures like B-trees to higher dimensions [24]. Each node in an R-tree encloses a bounding rectangle that contains all its children (fig. 9a).

One approach to implementing R-trees is to impose a linear ordering (using a space-filling curve to preserve locality) on two-dimensional keys and store nodes in a one-dimensional structure like a B-tree [25, 26]. Each node in our R-tree is a record that encodes the 2D coordinates of the rectangle’s corners along with a Z-order transform [27, 28]. To bulk load records into an R-tree, we transform mapped coordinates to the Z-order curve and sort on the Z-value. To build each level of the tree’s internal nodes, a streaming reduction pass uses an accumulator to track children’s bounds.

Window query Window queries on an R-tree find all leaves that intersect a search rectangle and are analogous to B-tree range queries. R-trees allow overlapping rectangles in inner nodes, so unlike B-trees, search paths through the tree may diverge taking multiple paths to leaf nodes. To account for limited queue size in scratchpads, we spill search threads to a queue in DRAM. We parallelize window queries across multiple streams by splitting up the search rectangle and performing multiple smaller window queries in parallel.

Spatial join A spatial join between two tables concatenates pairs of geographic objects that satisfy a spatial predicate (e.g., distance, intersection, etc.). For example, ridesharing applications use spatial joins to pair riders with nearby drivers or identify recent demand in a region by joining users’ GPS tracks with city boundaries. There are many spatial join algorithms [29, 30], each optimized for different queries and indices. Figure 9b shows a nested loop join between two indices that checks for overlapping nodes.

V. Evaluation

We quantify our design’s area overhead and evaluate Aurochs using cycle-accurate simulation.

A. Area overhead

Aurochs extends Gorgon’s SRAM scratchpad with a low-overhead memory allocation pipeline [8]. We implement these micro-architectural additions in Chisel [31] and synthesize the RTL with Synopsys Design Compiler [32] and a 15 nm predictive PDK [33]. The 15 nm PDK does not have an SRAM compiler, we use latch-based arrays for small memories and scale large memories from the industrial 28 nm PDK used by Gorgon [3]. The design meets timing at 1 GHz with the critical path from the issue queue through the allocator.

Figure 10 shows the overhead and per-component breakdown of the design as a percentage of Gorgon’s baseline scratchpad area. Aurochs increases scratchpad area by 15% which corresponds to a 5% total area increase. Notably, the allocation logic that schedules requests out-of-order by matching lanes and banks occupies only a small portion [8].

B. Simulation

Aurochs lowers a manually-planned SQL operator tree to a graph of compute and scratchpad tiles. Query planning is a well-studied topic, so we do not consider it in this paper. Nodes in the tree represent physical operators in a SQL query plan but with parallelization parameters to trade off throughput with
compute and scratchpad tile requirements. A custom place and route tool maps these tiles onto the accelerator fabric to account for the on-chip interconnect’s latency and bandwidth.

We evaluate our design’s end-to-end performance with a custom cycle-accurate simulator written in C++; our simulator incorporates Ramulator [34] for cycle-accurate HBM simulation. We compare Aurochs’ simulated performance against CPU and GPU baselines outlined in Table 1. The multi-socket server runs a time-series database [35, 36] with extensions for geospatial queries [37] and machine learning [2, 38]. The GPU runs queries written with CUDA libraries [15] from NVIDIA for database, geospatial, and machine learning processing on a single in-memory table format. Our GPU has approximately a terabyte per second of DRAM bandwidth but limited 16 GiB capacity. For a fair comparison, we pre-load tables into the GPU’s memory and profile only kernel runtimes.

Scaling datasets Figure 11 shows how kernels’ performance scale with increasing table sizes compared to the baseline Gorgon using suboptimal algorithms. Cycle-accurate simulation imposes practical limits on table sizes, so we project performance at larger datasets using an analytical model validated against smaller cycle-level simulations. Aurochs outperforms Gorgon at large table sizes due to better asymptotic performance. Sort-merge joins use dense access patterns that conserve memory bandwidth, so they run faster than hash joins at small sizes despite sort runtime scaling super-linearly with table size. When table sizes grow, hash joins begin to outperform sort-merge joins. Spatial joins in particular are infeasible on Gorgon because without spatial indices, a spatial join requires all-to-all comparisons using a nested loop join, making it impractical for real-world datasets. Figure 11b shows spatial join performance between a fixed size table as the other table scales. Gorgon presorts the larger table which causes $O(n \log n)$ asymptotic behavior while Aurochs searches a spatial index with $O(\log n)$ time.

Aurochs matches software’s asymptotic performance but outperforms both the CPU and GPU at all table sizes due to better constant factors. The GPU joins two tables each with 100 million 8 byte tuples at 4.5 GB/s, outperforming the CPU by over an order of magnitude. When parallelized, Aurochs can join tables at over 50 GB/s. The difference in spatial join performance between the designs is similar.

Scaling throughput Our architecture’s regular structure of compute and scratchpad tiles enables scaling database kernels to trade off throughput with resource requirements. All our database kernels map across parallel vector streams at the cost of additional compute and scratchpad tiles. Figure 12 shows that kernels’ performance scales with increasing stream-level parallelization; kernels become memory-bound when sufficiently unrolled, even at terabyte-per-second memory bandwidths. However, the observed throughput is far below DRAM bandwidth due to super-linear algorithms or sparse access patterns.

Benchmark queries Existing database benchmarks do not reflect the characteristics of emerging analytics workloads for which Aurochs is built. Analytics is gravitating towards real-time stream processing [39] and often uses shallow ml models to identify latent variables with low latency. So, to evaluate end-to-end query performance, we write our own queries shown in fig. 13 and explained in table 2. The business case for these queries is a ridesharing service that runs real-time analytics to streamline its operations. The query data is synthetic; queries and their generators are available*. Table 2 outlines table sizes and schema; ride request and driver status streams join against fact tables holding historical data. The GPU baseline does not support stream-processing, so we treat the stream tables as materialized tables with pre-built indices. Ridesharing services price locations in real-time based on driver-rider imbalance or predict trip durations by querying for historical trips with similar metrics. For example, q3 uses geospatial indices to gauge demand by querying for rider activity per-location over a recent interval.

Figure 14 shows absolute runtime and energy (estimated by multiplying runtime with design power) between Aurochs and both baselines. Aurochs outperforms the GPU on all queries by up to $12 \times$. And, our design on average outperforms the CPU by $160 \times$ and GPU by $8 \times$ (fig. 14). When accounting for design power, Aurochs is more energy efficient than a GPU by $20 \times$ and GPU by $8 \times$ (fig. 14).

VI. Related Work

We summarize related work in database acceleration and architectures and techniques for irregular algorithms, describing the differences from Aurochs.

Dataflow accelerators Other dataflow architectures accelerate database kernels. Dadu et al. [40] propose the spu, a dataflow accelerator that targets general-purpose sparse workloads. The design outperforms a software database by about $10 \times$ and supports hash joins if tables are small enough to fit on-chip; it

* github.com/stanford-ppl/rideshare
Architectures for irregular parallelism

Swarm [43]–[45] is tiled architecture of simple cores with hardware support for fine-grained tasks. A swarm task is a short program, often just tens of instructions, but tasks themselves are not known until runtime and may conditionally spawn child tasks. Swarm speculatively executes tasks out-of-order, detecting and aborting tasks with dependence violations. By contrast, our design does not speculate, and threads do not have ordering dependencies. Consequently, Swarm’s programming model is less restrictive than our threading model, and speculation can discover parallelism that would be impossible to extract otherwise. However, when kernels can be expressed with our threads, Aurochs simpler, more regular design avoids speculation’s complexity in hardware and provides higher throughput by keeping more threads in flight.

Emu [46]–[48] is an architecture that uses lightweight thread contexts similar to our records to migrate threads to the data they reference. As in our design, threads can also spawn new threads at runtime which will migrate to the referenced memory without software intervention. Emu’s focus is scale-out performance; it migrates thread state automatically between nodes to maintain locality even in applications with irregularity. Instead, our focus is better per-node performance on irregular workloads which we accomplish by keeping enough thread contexts active to fully capitalize on dataflow architectures’ deep pipelines. Our design also emphasizes bandwidth efficiency by partitioning problems to fit on-chip and scheduling threads to maximize SRAM bandwidth.

Accelerating irregular algorithms

Existing techniques to accelerate irregular algorithms fall under several categories: microarchitectural extensions to existing architectures or improved code-generation, new fixed-function accelerators, or overlay architectures for FPGAs. Our design uses similar principles as these works to maximize performance by keeping compute lanes active, but we avoid the shortfalls of inefficient hardware that is too general-purpose or fixed-function hardware that is too inflexible.

To better support irregularity, several works propose extensions to existing architectures or better code-generation techniques. ElasticFlow [49] pipelines irregular loop nests on FPGAs; when inner-loops are variable-length and have loop-carried dependences, it runs multiple outer iterations in parallel and out of order. Vaidya et al. [13] propose microarchitectural techniques to improve SIMD efficiency on GPUs. Srinath et al. [50] propose ISA extensions that encode interiteration loop dependence patterns; explicitly annotating these dependencies decouples iteration ordering like in our design. Lang [51] propose code-generation techniques for database kernels to better cope with branch divergence when compiling queries using SIMD.

Other works accelerate database kernels with either FPGAs or fixed-function accelerators. Koberber et al. [52] implement a fixed-function accelerator for database hash indices. Kumar et al. [53] propose fixed-function hardware for software data structures. Sun et al. [54] offload LSM tree compaction on an FPGA. Halstead et al. [55] keep many concurrent threads
SELECT COUNT(*) AS rideCount FROM rideReq AS req JOIN driverStatus AS ds ON GEO.DIST(ds.pos, req.start, 1 km) INNER JOIN driver AS d ON d.driverId = ds.driverId WHERE req.seats = d.seats AND s.time >= NOW - 5 days GROUP BY s.driverId

**Query 2**

SELECT COUNT(*) AS rideCount FROM location AS l JOIN rideReq AS r ON GEO.DIST(l.bounds, r.start) WHERE l.locationId = 0 GROUP BY INTERVAL(time, '10 min') ORDER BY rideCount

**Query 3**

SELECT COUNT(*) AS rideCount FROM location AS l JOIN rideReq AS r ON GEO.DIST(l.bounds, r.start) WHERE r.time > NOW - 1 minutes GROUP BY locationId ORDER BY rideCount

**Query 4**

SELECT req, riderId, o, ..., c11 FROM (SELECT r.* FROM ride AS r JOIN location AS l ON GEO.DIST(l.bounds, r.start) WHERE l.locationId = 0 AND r.starttime > NOW - 5 days) AS req, ..., c11) AS kmInf FROM rideReq AS req JOIN location AS l ON GEO.DIST(l.bounds, req.start) WHERE l.locationId = 0 AND req.driverId = 0

**Query 5**

SELECT *, d.driverId, AVG(s.cols) OVER w AS c0, ..., MAX(s.cols) OVER w AS c11, SYN.PREDICT(model, [c0, ..., c11]) FROM driverStatus AS ds JOIN driver AS d ON d.driverId = ds.driverId WINDOW w AS ( PARTITION BY s.driverId ORDER BY size RANGE.BORDERS(2) )

**Query 6**

SELECT *, SYN.PREDICT(model, [c0, ..., c11]) FROM (SELECT locationId, COUNT(*) AS r FROM location AS l JOIN rideReq AS r ON GEO.DIST(l.bounds, r.start) GROUP BY locationId ) AS r JOIN ( SELECT locationId, COUNT(*) AS c FROM location AS l JOIN driverStatus AS d ON GEO.DIST(l.bounds, d.pos) GROUP BY locationId ) AS d ON r.locationId = d.locationId JOIN location AS l ON l.locationId = r.locationId

**Query 7**

SELECT *, r.riderId, AVG(d.rating) AS c0, ..., AVG(c11) AS c11, LOG.REG.PREDICT(model, [c0, ..., c11]) FROM ride JOIN rider AS r ON ride.riderId = r.riderId JOIN driver AS d ON ride.driverId = d.driverId WHERE ride.starttime > NOW - 30 days GROUP BY r.riderId

**Query 8**

SELECT *, r.riderId, AVG(ride.cols) AS c0, ..., AVG(ride.cols) AS c11, KMEANS_INTER(model, [c0, ..., c11]) FROM ride JOIN rider AS r ON ride.riderId = r.riderId JOIN location AS l ON GEO.DIST(l.bounds, ride.start) WHERE l.locationId = 0 GROUP BY r.riderId

**Query 9**

SELECT d.*, GEO.DIST(r.start, d.pos) AS dist FROM driverStatus AS d JOIN rideReq AS r ON GEO.DIST(r.start, d.pos, 1 km) WHERE r.riderId = 0 ORDER BY dist LIMIT 100

**Fig. 13** A ridesharing benchmark query set that runs end-to-end analytics on geospatial and time-series data

**Table 2** Descriptions of our benchmark queries and their input table sizes

<table>
<thead>
<tr>
<th>Table rows</th>
<th>Stream rows</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10K</td>
<td>-</td>
</tr>
<tr>
<td>Q2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Q3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Q4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Q5</td>
<td>10K</td>
<td>-</td>
</tr>
<tr>
<td>Q6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Q7</td>
<td>1M</td>
<td>1M</td>
</tr>
<tr>
<td>Q8</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>Q9</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
active when building and probing hash tables on FPGAs to mask thread latency. Finally, István et al. [56] implement a line-rate hash table for FPGAs.

Prior ASICs for data structures only accelerate a limited set of kernels that do not share common hardware. Fixed-function designs can be more power-efficient than RDAs, but data center operators avoid them due to lack of programmability and high cost [9, 57]. RDAs provide coarser primitives than FPGAs that better match application domains, providing higher performance at lower power without compromising programmability [7]. Aurochs is a unified database architecture that accelerates SQL but also data structures by repurposing the same record-processing hardware. Unlike these designs, our architecture’s regularity enables parallelizing across many compute and memory tiles—without global scratchpads or synchronization hotspots—to scale to terabyte-per-second memory bandwidths. Multi-purpose accelerators better justify the cost of supporting resources like large scratchpads and high-bandwidth memory.

VII. CONCLUSION

Aurochs shows that dataflow architectures can run irregular algorithms at line rate and match CPUs’ asymptotic performance with just a simple microarchitectural extension. Lifting thread state out of CPUs’ register files and into the dataflow decouples threads from a warp or lane, keeping compute units active even during branch divergence. Limiting cross-thread communication to atomics enables our architecture to keep spatially distributed pipelines full even with long network delays and allows aggressive reordering to minimize bank conflicts.

Most importantly, our implementation is low-overhead and piggybacks on the vector record-processing hardware already native to database accelerators. Application domains less pervasive than SQL cannot afford high performance’s cost: large dice in advanced process nodes with high-bandwidth memory. Data center operators want flexible hardware and cannot afford to deploy fixed-function designs that limit themselves to a single kernel. We show the potential to accelerate an entire class of algorithms with irregular parallelism from a single dataflow accelerator.

<table>
<thead>
<tr>
<th>Runtime (ms)</th>
<th>Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>GPU (ms)</td>
</tr>
<tr>
<td>Q1</td>
<td>16</td>
</tr>
<tr>
<td>Q2</td>
<td>7</td>
</tr>
<tr>
<td>Q3</td>
<td>8</td>
</tr>
<tr>
<td>Q4</td>
<td>41</td>
</tr>
<tr>
<td>Q5</td>
<td>18</td>
</tr>
<tr>
<td>Q6</td>
<td>23</td>
</tr>
<tr>
<td>Q7</td>
<td>21</td>
</tr>
<tr>
<td>Q8</td>
<td>6</td>
</tr>
<tr>
<td>Q9</td>
<td>3</td>
</tr>
</tbody>
</table>

![Fig. 14 Baseline query performance comparison and average speedup](image-url)

REFERENCES


