

# Evaluating Apple Silicon for Data Processing

Alexander Beischl\*  
Technical University of Munich  
Munich, Germany  
beischl@in.tum.de

Mykola Morozov\*  
Technical University of Munich  
Munich, Germany  
morozov@in.tum.de

Michael Jungmair  
Technical University of Munich  
Munich, Germany  
jungmair@in.tum.de

Lukas Haussmann  
Technical University of Munich  
Munich, Germany  
lukas.haussmann@in.tum.de

Thomas Neumann  
Technical University of Munich  
Munich, Germany  
neumann@in.tum.de

## Abstract

Apple Silicon’s M-Series integrates powerful GPUs with a physically unified memory architecture, which makes it a promising platform for the growing shift toward local-first analytical data processing on consumer devices. However, its suitability for database workloads remains largely unexplored.

In this paper, we present the first database-oriented study of Apple Silicon. We identify synchronization costs incurred by using shared memory for CPU and GPU, and propose an approach to reduce this overhead to a one-time cost by explicitly managing memory in the database system. Further, we analyze the Metal programming model for database workloads. While it enables efficient GPU execution, it imposes non-trivial synchronization constraints and requires careful kernel structuring. Finally, we evaluate the architecture using end-to-end analytical query prototypes that integrate our findings on memory management, kernel structuring, and synchronization. Our results demonstrate that GPU execution and CPU-GPU co-processing on Apple Silicon can achieve speedups of up to 5× over CPU-only execution and can be leveraged to accelerate local-first data analytics.

## CCS Concepts

• **Information systems** → **Database query processing**.

## Keywords

Apple Silicon, heterogeneous co-processing, OLAP, local-first

### ACM Reference Format:

Alexander Beischl, Mykola Morozov, Michael Jungmair, Lukas Haussmann, and Thomas Neumann. 2026. Evaluating Apple Silicon for Data Processing. In *22nd International Workshop on Data Management on New Hardware (DaMoN ’26)*, May 31–June 05, 2026, Bengaluru, India. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3789237.3809127>

## 1 Introduction

Increasingly, data is also processed on users’ personal machines rather than on centralized servers, especially for workloads that fit

\*Both authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution 4.0 International License. *DaMoN ’26, Bengaluru, India*

© 2026 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-2455-8/26/05  
<https://doi.org/10.1145/3789237.3809127>

within consumer hardware constraints. This trend towards local-first data processing is also reflected in the widespread adoption of embedded analytical systems like DuckDB, which is primarily used on consumer devices [59]. At the same time, consumer devices have been equipped with increasingly powerful GPUs to better support machine learning workloads. In particular, Apple devices such as MacBooks are widely deployed [55] and integrate powerful on-chip GPUs with unified memory architectures, which makes them promising for database workloads.

Motivated by this shift, prior work has investigated accelerating database workloads on consumer-grade hardware using discrete GPUs [21, 29, 44, 68], and integrated GPUs [20, 48, 69]. The majority of these studies focus on architectures with separate memory domains. In contrast, only a small number explicitly explore unified memory systems, such as APUs [12, 27, 28], which were historically not computationally powerful.

However, we are not aware of research on data processing on Apple Silicon, even though its architecture is particularly promising for analytical database workloads, which are often memory-bound. Its system-on-a-chip design integrates heterogeneous CPU cores, a high-throughput GPU, specialized accelerators, and a shared, unified memory with advertised bandwidths of several hundred GB/s. Prior work has primarily studied Apple Silicon in the context of LLM training [13, 19, 60], LLM workload interference [5, 7–9], scientific- [32], and high-performance computing [30].

In this paper, we present the first systematic study of Apple Silicon from a database systems perspective, comprising three main contributions. First, we characterize the performance of its unified memory architecture and identify an approach to avoid common overheads when sharing memory across CPU and GPU. Second, we analyze the GPU’s Programming Model (Metal) in the context of database workloads and present several surprising insights. Finally, we implement relational analytical query prototypes for CPU, GPU, and CPU-GPU co-processing, and use them to perform an end-to-end evaluation of this hardware platform.

We first motivate in more detail why Apple Silicon is an attractive platform for data processing workloads and provide an overview of its architecture in section 2. Next, in section 3, we measure achievable bandwidth and latency using micro-benchmarks and show that, by default, there is significant overhead when allocating shared memory. To avoid this overhead, and thus make GPU data processing viable, we propose managing memory directly in the database system. In section 4, we analyze Metal, the GPU programming model, in detail and find that synchronization, often required

for database workloads, is not straightforward. To tackle this, we contribute Metal-specific spin locks. Based on the insights from section 3 and section 4, we design analytical query prototypes for simplified SSB queries using either the CPU, GPU, or co-processing. Our results show that GPU execution can accelerate query execution by up to 5 $\times$  over CPU execution, while co-processing can often further speed up execution workloads. Finally, we conclude with a discussion of related work.

## 2 Motivation

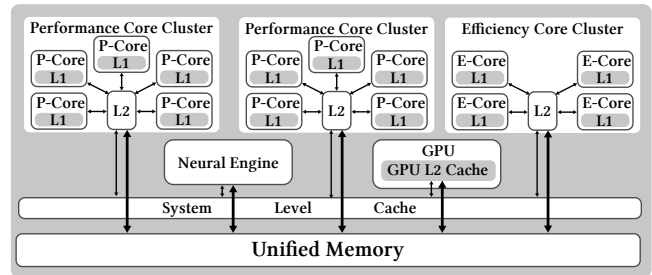
Apple Silicon’s M-series has become a widely deployed hardware platform in consumer-grade laptops. Its system-on-a-chip (SoC) integrates multiple CPU cores, an integrated GPU, accelerators, such as a Neural Engine, and a high-bandwidth memory subsystem. The CPU follows ARM’s *big.LITTLE* architecture, combining performance and efficiency cores (cf. Figure 1). In addition, the integrated GPU provides substantial parallel compute throughput, extending the platform beyond traditional CPU-centric execution.

Together, these characteristics make Apple–Silicon–based systems interesting for executing non-trivial data-processing workloads. At the same time, the growing importance of local and local-first data processing [59], combined with the widespread use of MacBooks among developers and researchers, raises the question of how Apple Silicon’s architectural characteristics can be leveraged by database management systems (DBMSs).

All processing units are directly connected to a unified memory subsystem, rather than operating on separate device-local memory, and share a system-level cache [66]. This design removes the explicit separation between CPU and GPU memory spaces, allowing all processing units to access a single physical memory pool. The unified memory provides high theoretical memory bandwidth for a consumer-grade platform, with advertised peak bandwidths ranging from approximately 120 GB/s to over 800 GB/s, depending on the chip variant. Such bandwidth levels are particularly relevant for database workloads, as many (especially OLAP) workloads are primarily limited by memory throughput rather than computation. However, it is not specified to what extent these advertised bandwidths are achievable in practice and whether the effective memory bandwidth differs between the CPU and the GPU.

Beyond high memory bandwidth, the unified memory architecture, in principle, eliminates the need for explicit data transfers between CPU and GPU memory spaces, as both processing units operate on the same physical memory. It remains an open question whether working on shared memory incurs overhead for synchronizing access and modifications made by the CPU and GPU.

Taken together, Apple Silicon’s combination of heterogeneous compute resources, high advertised memory bandwidth, and a physically unified memory architecture promises benefits for data-intensive workloads. However, it is unclear how much of the available memory bandwidth can be effectively utilized by CPU and GPU, what overheads arise from shared memory and coherence mechanisms, and how well GPU execution or CPU-GPU co-processing via Metal’s programming model aligns with the requirements and execution patterns of database workloads.



**Figure 1: Schematic overview of the Apple M4 Pro SoC, illustrating how heterogeneous compute resources are connected to unified memory and caches.**

To address these open questions, we conduct a systematic experimental evaluation of Apple Silicon on two systems: a MacBook Pro equipped with an M4 Pro (10 performance, 4 efficiency CPU cores, 20 GPU cores, 48 GB LPDDR5X-8533 Unified Memory, macOS 26.4.1, 2799\$) and a Mac Studio equipped with an M3 Ultra (24 performance, 8 efficiency CPU cores, 80 GPU cores, 256 GB LPDDR5-6400 Unified Memory, macOS 15.5, 7899\$). The M4 Pro is a single-die SoC, while the M3 Ultra SoC consists of two M3 Max dies connected via Apple’s UltraFusion interconnect.

## 3 Unified Memory

Apple Silicon employs a unified memory architecture in which all processing units share a single coherent memory pool connected via high-bandwidth on-chip interconnects. This design promises high memory throughput and, in principle, eliminates the need for explicit data transfers between CPU and GPU, as both can operate on the same physical data. Such properties are particularly appealing for database systems, whose workloads are primarily limited by memory throughput rather than computation.

Thus, in this section, we measure the achievable unified-memory bandwidth for CPU and GPU and evaluate whether shared memory eliminates transfer costs or rather shifts them to memory-coherence overhead for synchronizing modifications by CPU and GPU

### 3.1 CPU & GPU Memory Bandwidth

Apple reports high theoretical unified memory bandwidths for consumer-grade devices, but does not specify the attainable memory bandwidth for CPU and GPU. We therefore evaluate the achievable CPU memory bandwidth and latency, and compare them to the GPU memory bandwidth to assess the opportunities of both processing units for database workloads.

We implement multi-threaded sequential read and write benchmarks to measure the achievable CPU memory bandwidth, while memory latency is evaluated through pointer chasing with random accesses uniformly distributed across the dataset. All benchmarks operate on a 24 GB dataset, partitioned into one morsel per thread (one thread per core) to minimize cache effects and ensure memory-bound execution. We vary the number of threads from one up to the total number of CPU cores. GPU memory bandwidth is measured using a kernel that reads each tuple of a large array and modifies its value to prevent compiler optimizations from eliminating memory

**Table 1: Benchmark results for different Apple Silicon Macs.**

Memory Bandwidth	M4 Pro	M3 Ultra
Theoretical	273GB/s	819.2GB/s
GPU (Total)	248GB/s	738GB/s
CPU (Total)	251GB/s	253GB/s
CPU (Single Core)	52GB/s	43GB/s
CPU Latency	20ns	21.7ns

accesses. We evaluate data sizes ranging from 8–24 GB on the M4 Pro and 8–196 GB on the M3 Ultra. Each experiment is repeated ten times, and we report the median of the measured values.

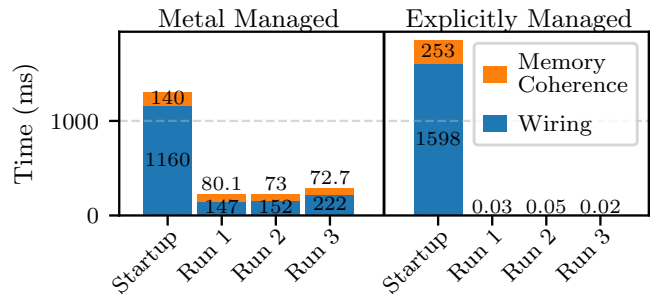
Our results show low CPU memory latencies of  $\approx 20ns$  on both systems (cf. Table 1). On the M4 Pro, both the CPU and GPU achieve up to 90% of the advertised memory bandwidth, suggesting that differences in query execution performance between the CPU and GPU are primarily influenced by compute capabilities and programming model rather than memory throughput. In contrast, on the M3 Ultra, the GPU attains 90% (738 GB/s), while the CPU reaches only 30% (253 GB/s). Notably, despite having more than twice as many CPU cores (32 vs. 14), the M3 Ultra’s CPU achieves a similar memory throughput to the M4 Pro. This observation indicates that the CPU-side interconnect on the M3 Ultra limits memory throughput and prevents the CPU from fully exploiting the available unified memory bandwidth. Consequently, for memory-bound workloads, the GPU is the more suitable processing unit on M3 Ultra.

### 3.2 Unified Memory Coherence

A central question is how unified memory affects data movement costs between CPU and GPU. By allowing both processors to operate on the same physical memory, unified memory can eliminate explicit data transfers altogether. However, these costs may not disappear but instead be shifted to memory coherence mechanisms that synchronize modifications between processors, i.e., flushing device caches to memory to ensure that CPU modifications are visible to the GPU, and vice versa. In this section, we analyze the overhead of fundamental coherence mechanisms by the system that cannot be eliminated through careful engineering, while we discuss memory coherence management by the developer in subsection 4.3.

System-induced memory coherence overhead arises when allocating CPU- and GPU-shared memory via the Metal API. Before launching GPU kernels, the system must ensure that all memory accessed by the GPU is coherent, i.e., that all modified (dirty) cache lines are synchronized. We observe that this coherence pass is triggered at kernel launch for every newly allocated shared memory buffer. Therefore, the system initializes a fresh buffer (called *wiring*) at a throughput of 10–60 GB/s, followed by a coherence pass over the entire buffer with 70–120 GB/s on M4 Pro.

We illustrate the implications using a micro benchmark in Figure 2 (Metal Managed). At startup, we allocate 16 GB of shared memory via the Metal API to load the dataset, triggering wiring (1160ms) and memory coherence (140ms). Then, for each micro benchmark run, we allocate additional 8 GB of working memory via the Metal API, again incurring full wiring and coherence passes.

**Figure 2: Explicitly managing memory avoids significant overheads compared to Metal.**

As a result, wiring and coherence costs can dominate workload execution time, particularly for memory-intensive workloads.

A key contribution of this work is the insight that wiring and coherence overhead are not inherent costs of unified memory but are caused by buffer allocation through the Metal API. To avoid these repeated passes, we instead allocate a large shared memory region (24 GB) once at startup and manage it explicitly within the DBMS, thereby bypassing Metal API buffer allocations. Since the underlying memory persists across executions, a full wiring and coherence pass is triggered only once. Thereafter, only modified regions are continuously synchronized. This approach reduces coherence overhead to less than 0.06ms per run, as shown in Figure 2 (Explicitly Managed) and aligns with existing database systems, which often manage memory explicitly through buffer managers to control placement, lifetime, and reuse [17, 18, 34, 40, 42, 45].

An additional benefit of our approach is explicit memory lifetime management: Metal does not immediately release memory but instead frees resources gradually, which can lead to excessive memory consumption if buffers are not managed explicitly [19].

Overall, these results show that while unified memory eliminates explicit data copies, coherence enforcement remains a non-negligible cost. With appropriate memory management, this cost can be effectively amortized, reducing coherence overhead to a one-time cost for analytical workloads.

## 4 The Metal Programming Model for Data Processing

Apple Silicon GPUs are programmed using the Metal Shading Language. This section analyzes Metal’s programming model in the context of database workloads and presents several surprising insights into resource management and control-flow synchronization.

### 4.1 Scheduling Computations on the GPU

Running hardware-accelerated computations with the Metal API requires creating a compute pass for a specific GPU device [52]. Listing 1 shows a compute pass consisting of one or more compute encoders dispatching a specified number of GPU threads to execute kernel functions with given parameters. These kernel functions can be loaded from textual Metal shaders or precompiled binaries  $\bullet$ . The parameters include objects such as memory buffers and

**Listing 1: Simplified Compute Pass Setup**

```

1 auto *dev = MTL::CreateSystemDefaultDevice();
2 auto *lib = dev->newDefaultLibrary(); ❶
3 auto *func = lib->newFunction("my_kernel");
4 auto *s = dev->newComputePipelineState(function, nullptr);
5 auto *cq = dev->newCommandQueue();
6 auto *cb = cq->commandBufferWithUnretainedReferences(); ❷
7 auto *enc = cb->computeCommandEncoder();
8 enc->setComputePipelineState(state);
9 // Specify resources at specific kernel parameter indices
10 enc->setBuffer(data, /* offset */ 0, /* atIndex */ 0);
11 enc->dispatchThreads(gridSize, threadGroupSize); ❸
12 enc->endEncoding();
13 cb->commit(); ❹
14 cb->waitUntilCompleted();

```

rendering resources that need to be bound within a compute encoder. Metal tracks and retains object references used in hardware-accelerated passes to ensure memory safety, unless this behavior is manually disabled to reduce CPU overhead ❷.

Kernel execution in a command encoding is dispatched across a grid of threads on one or more GPU cores, which are divided into threadgroups and executed together ❸. At a low level, threads execute in lockstep using the Single-Instruction-Multiple-Threads (SIMT) model, which Apple refers to as SIMD groups, within a threadgroup. Metal includes additional lockstep-based operations for transferring and synchronizing memory between specific threads inside the SIMD group.

The common approach of overdispatching threads on the grid to hide memory and instruction latency also applies to Apple Silicon GPUs [61]. Non-aligned thread groups and grid sizes result in under-utilized SIMD groups with idle threads. After specifying the resource references and dispatching threads, the command encoding is finalized. Then the command buffer can be committed, which launches the kernel asynchronously on the specified GPU device ❹. To ensure execution completes and the memory used in the compute pass is coherent, a CPU thread can synchronously wait for the command buffer to complete.

To verify previously published data [19], we measured the kernel startup time, consisting of the encode phase—*the CPU scheduling the command buffer*—and the dispatch phase—*the GPU preparing the dispatch of the first portion of scheduled threads*. The encode phase took 28 – 79µs on all machines, while the dispatch phase depends on the underlying hardware performance, taking 14-17 µs on M4 Pro and 65-75 µs on M3 Ultra.

## 4.2 Memory Address Spaces and Access

Just like programming models for other GPU vendors, Metal offers multiple address spaces for the memory resources used inside kernels [6]. threadgroup memory is shared between all threads executed within a threadgroup, while the constant (read-only) device (read-write) address spaces are shared across the entire GPU device. Since separate SIMD groups and threadgroups can be scheduled on different GPU cores having their own cache, the developer is responsible for synchronizing data access, control flow, and memory write coherency.

While the Metal Shading Language provides threadgroup-level memory and control-flow synchronization functions, GPU-wide unified memory coherency requires explicit opt-in. Metal allows specifying the scope of coherency for memory operations, i.e., which threads observe the result of a store after synchronization: the writing thread, the owning threadgroup, or the whole device. Device-level memory coherency can be forced by using the qualifier `coherent(device)`, which ensures that the writes to unified memory are completed by flushing the caches. We have not observed a noticeable slowdown from using device-level memory coherency.

## 4.3 Synchronization

Apple Silicon provides standard synchronization primitives, including intra-group shuffles for exchanging data within a SIMD group. However, atomic operations are only fully supported for up to 32 bits, which makes explicit locking necessary in many scenarios, e.g., parallelized hash table construction or merging SIMD group aggregation results.

Locking in Metal is non-trivial. Unlike many modern GPUs, which provide per-thread scheduling resources, e.g., program counters and call stacks [41], Apple Silicon executes SIMD groups in true lockstep. As a consequence, conventional spin locks do not work, as the SIMD group can only make progress once all threads have acquired the lock. This immediately leads to deadlocks when two threads in a SIMD group contend for the same lock, or when different SIMD groups contend for overlapping lock sets, and each group only acquires some of the locks. We thus introduce Metal-specific spin locks (cf. Listing 2) with different performance trade-offs.

*Serial spin locks* avoid deadlocks within SIMD groups by serializing lock acquisition. Only the first active thread in a SIMD group can request its lock, perform its work, and release the lock (lines 3-10). Afterward, it is the next active thread's turn. Serializing locking attempts prevents deadlocks, but at the cost of underutilizing the SIMD group, since only one thread performs useful work.

Our second approach uses *non-blocking spin locks*. Each thread attempts to acquire its lock using `try_acquire`. If successful, it performs its work, releases the lock, and terminates (lines 14-20). Otherwise, it retries in the next iteration. Thus, deadlocks are prevented by successful threads progressing while unsuccessful threads busy-wait and retry later. However, contention for shared locks is higher than in the serial approach, as all threads try to acquire their locks at the same time.

Serial and non-blocking spin locks both prevent deadlocks caused by Apple Silicon GPUs' true lockstep execution. However, neither technique utilizes the SIMD group optimally, either because execution is serialized or because busy-waiting increases lock contention. Therefore, we extend both approaches with group-wise locking.

*Serial spin locks with grouping* group all threads within a SIMD group that target the same lock. Groups are processed sequentially, and the first thread of each group (leader) requests the lock on behalf of its group. Once the lock is acquired, all group members perform their work serially under this single lock, and eventually, the leader releases it. This reduces the number of lock acquisitions from one per thread to one per lock group. However, progress within a SIMD group is still limited to one lock group at a time.

Listing 2: Per-Thread Metal Spin Lock

```

1  /* Serial spin lock */
2  auto* lockPtr; // Lock the thread wants to acquire
3  while (true) {
4      if (first_in_simd_group()) {
5          spinlock_acquire(lockPtr);
6          doWork();
7          spinlock_release(lockPtr);
8          break;
9      } // else: spin until first
10 }
11
12 /* Non-blocking spin lock */
13 auto* lockPtr; // Lock the thread wants to acquire
14 while (true) {
15     if (spinlock_try_acquire(lockPtr)) {
16         doWork();
17         spinlock_release(lockPtr);
18         break;
19     }
20 }

```

Alternatively, *non-blocking spin locks with grouping* combine non-blocking locks with the idea of grouping threads targeting the same lock. Within each group, threads are serialized, and only the first thread (leader) tries to acquire the lock. On success, the leader holds the lock while all group members perform their work in serial order. Unlike *serial spin locks with grouping*, the leaders of all lock groups can attempt to acquire their locks concurrently, allowing multiple independent lock groups to make progress within the same SIMD group.

The different variants trade off lock contention, grouping overhead, and SIMD group utilization differently. We therefore evaluate them by varying the number of available locks, which controls how frequently threads in a SIMD group contend for the same lock (cf. Figure 3). For few locks, contention is high as many threads target the same lock. In this setting, *serial spin locks with grouping* perform best, as they coalesce threads targeting the same lock and acquire it only once per group. The additional serialization does not noticeably hurt performance, since the small number of locks already limits parallel progress. As the number of locks increases, contention decreases, and the benefit of grouping fades. For a higher number of locks, *non-blocking spin locks* perform best, as they avoid the overhead of forming lock groups while still allowing all threads whose lock acquisition succeeds to perform work. The grouped variants remain competitive, but their grouping overhead becomes unnecessary once lock conflicts are rare. Overall, *non-blocking spin locks with grouping* provide the most robust trade-off across the full range of lock counts. They reduce redundant lock acquisitions under high contention, while still allowing multiple independent lock groups within a SIMD group to make progress concurrently.

Beyond locks, we also observe that control flow in SIMD groups and threadgroups can be synchronized with barriers, while GPU-wide resources and execution flow can be synchronized by scheduling events and fences from the CPU side. Since GPU-wide synchronization can only be done between kernels on the CPU, read-after-write operations spanning the whole unified memory across

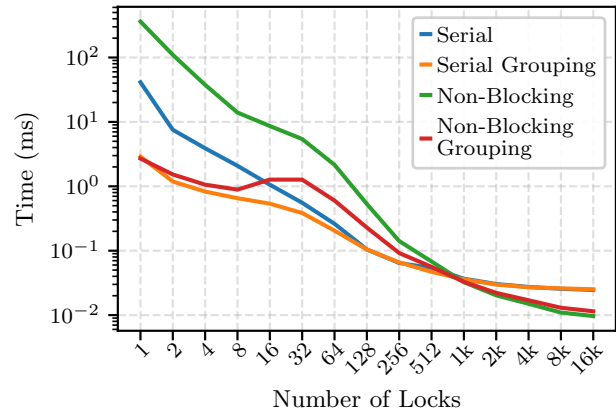


Figure 3: Performance of the different Metal-specific spin locks for different numbers of locks. Fewer locks cause more contention as more GPU threads request the same lock.

multiple threadgroups need to be split into separate kernels. This is particularly relevant for CPU-GPU co-processing, where shared state cannot be coordinated by GPU-side locking alone.

**Atomic Operations.** Atomic operations on Apple Silicon are only guaranteed at device level, i.e., atomics are consistent within CPU operations or within GPU operations, but not across devices. Although Apple Silicon synchronizes shared memory, modifications are not synchronized instantaneously. Thus, if the CPU and GPU access the same data in close succession, changes may not yet be coherent, even when using atomic operations.

We demonstrate this behavior with a stress-test microbenchmark. The benchmark initializes an array in shared memory with zeros and uses a shared atomic index that points to the next write position. Both CPU and GPU fetch a position from this index and increment it using `atomic_fetch_add`, before incrementing the corresponding array bucket. While most buckets are modified by only one device, 0.3-0.5% of the buckets are incremented twice, indicating that the index is not synchronized by atomic instructions of CPU and GPU, but rather the unified memory. Because atomic instructions do not work across devices, changes must be synchronized explicitly by splitting GPU execution at synchronization points.

## 5 End-to-End Query Processing

Building on the insights from section 3 and section 4, we evaluate end-to-end query processing on Apple Silicon. We first describe our query implementations and then use them to assess Apple Silicon’s suitability for end-to-end query processing.

### 5.1 Prototype Implementations

For the evaluation, we chose SSB simplified [47], which is widely used to test GPU implementations [10, 11, 14] due to its simplicity, saturating memory bandwidth, and offering heavy joins. The simplified version narrows down column data types of SSB, while preserving the original benchmark’s intent. We selected queries

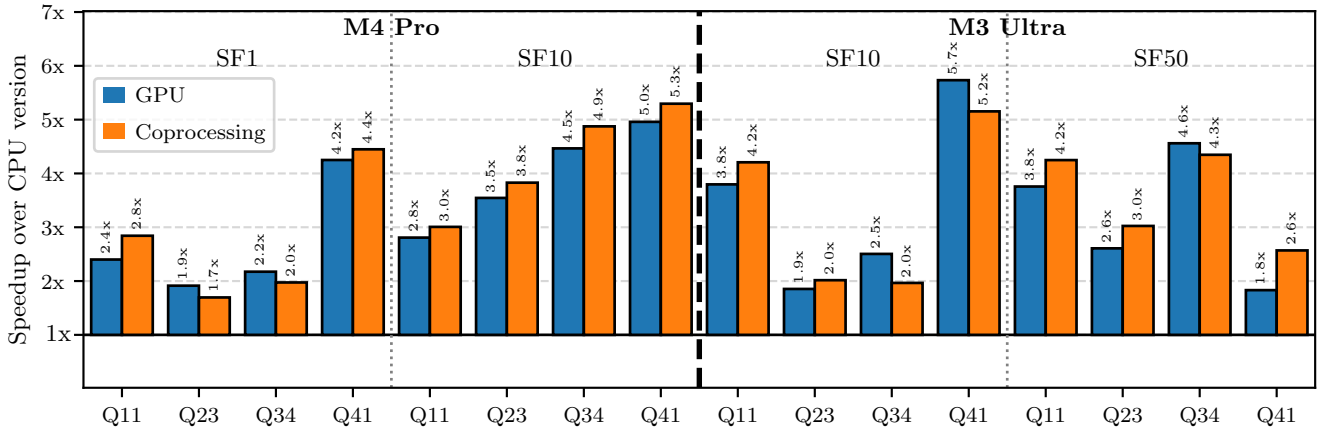


Figure 4: Speedup of DuckDB and our prototype’s GPU- and co-processing- versus the CPU-version for SSB simplified queries.

from four functional query groups, testing specific behavior of database engines for filtering and aggregation, multi-table joins, and complex grouping and aggregation.

Our implementation follows the code structure generated by compiling databases, applying operator fusion for data-centric query execution, and merging intermediate results only at materializing operators [39]. We ensure that our approach is generic with respect to data types by using an open-addressing linear-probing (OaLP) hash table for joins, and a partitioned chaining hash table for group-by pre-aggregation due to the GPU 32-bit atomics limits (cf. subsection 4.3). Our CPU prototypes use the OaLP hash table for join and group-by, achieving higher performance than the chaining hash table. Using the findings from subsection 3.2, we allocate memory and load data at system startup, manage memory ourselves, and clear intermediates and results once a query finishes.

On the *GPU*, we follow our insights from subsection 4.3 to over-allocate threads for dispatch and control the query execution flow. This involves splitting build and probe phases for joins and group-by into multiple kernels and synchronizing constructed structures by scheduling and triggering fences. The GPU merge step, implemented as a separate kernel, gathers per-thread and/or per-threadgroup results into the final output. On the *CPU*, the implementation saturates available threads, using barriers for synchronization points. We test the viability of *co-processing* on Apple Silicon for the selected queries. Therefore, we statically partition the input data between the CPU and GPU and run both implementations in parallel. Both processors build the index structures, e.g. hash tables, over the full dataset, while scans and probe-side work are restricted to their assigned data partitions. The physical memory ranges of allocated buffers are mapped to different addresses on CPU and GPU, hindering pointer usage for co-processing. Thus, we use relative offsets instead of pointers in our data structures while mimicking the Arrow data layout.

## 5.2 Evaluation

We test small- to large-sized workloads for consumer devices, running SSB scale factors 1, 10, and 50. We report end-to-end query

execution results for CPU, GPU, and co-processing separately on M3 Ultra and M4 Pro. Each query is repeated ten times, and we report the median. We also validated the performance of the CPU baseline against DuckDB [46], which is significantly slower due to the overhead of a full DBMS and using a vectorized execution model. Our experiments show that the GPU can achieve a speedup of 1.8–5.7× over the CPU on the same hardware platform, reading the same data from shared unified memory, and applying the same algorithms for query processing. We now analyze the results in more detail.

Q11 consists of a single table scan with three predicates, reducing the input by 50×, followed by an aggregation that sums the product of two columns. On the CPU, execution is straightforward but constrained by sequential data dependencies: the column for the next predicate is only accessed after all preceding predicates have qualified. Speculative execution can partly hide these dependencies. The GPU outperforms the CPU by 2.4–3.8×, exploiting higher parallelism to compensate for data dependencies and processing adjacent tuples within SIMD groups to leverage data locality. At the same time, the selective predicates cause thread divergence, many SIMD groups continue with only a few active lanes, and the GPU cannot fully saturate memory bandwidth. Thus, Q11 benefits from the GPU’s higher parallelism, but its high selectivity limits SIMD group utilization. While CPU execution time grows nearly linearly with data size, we observe GPU execution time grows slightly less than linearly. This difference stems from fixed kernel launch and scheduling overheads, which are more pronounced for small inputs and become increasingly amortized at larger scale factors.

For Q23 and Q34, the dominant pipeline consists of three consecutive hash-join probes. In Q23, the first probe reduces the input by 1000×, the second by 5×, and the final probe preserves all remaining rows. Q34 has three highly selective probes, each reducing its input by approximately 100×. Across all scale factors, the GPU outperforms the CPU by 1.9–3.5× for Q23 and by 2.2–4.6× for Q34. Performance is primarily determined by how efficiently each processor executes the probe pipeline over the memory-resident input tuples, since hash tables fit in the CPU and GPU caches across all

evaluated scale factors. CPU execution time scales largely with data size: on M4 Pro, execution time increases by  $\approx 7.5\times$  from SF1 to SF10, and on M3 Ultra, by  $\approx 4.3\times$  from SF10 to SF50. The times for the GPU scales less directly with input size, increasing by 3.8–4.0 $\times$  from SF1 to SF10 and respectively by 2.5–3.0 $\times$  from SF10 to SF50. This indicates that the GPU (20 cores on M4 Pro, 80 cores on M3 Ultra) requires sufficiently large data sizes to schedule its threads efficiently and fully utilize its compute capacities.

Q41 is dominated by four consecutive hash-join probes. The first two probes each reduce the input by 5 $\times$ , the third by 2.5 $\times$ , while the fourth preserves all remaining rows. Among the GPU executions, Q41 achieves the highest speedup over the CPU. This is due to its join selectivities: unlike Q23 and Q34, which eliminate most tuples in the first or second probe, Q41 maintains larger intermediate results throughout the pipeline. As a result, the probe pipeline is more prominent than in the other queries, and the GPU can exploit its high parallel compute throughput more optimally. This trend is also reflected in the scaling behavior. The CPU execution times scale as expected with the scale factors, increasing by 10.1 $\times$  and 5.4 $\times$ . On the M4 Pro, GPU execution scales similarly, increasing by 9.3 $\times$ , while still outperforming the CPU. On the M3 Ultra at SF50, however, GPU execution is significantly slower than expected, increasing by 17 $\times$  instead of the expected 5 $\times$ . This is caused by the larger hash tables in Q41 for SF50, which still easily fit into the CPU's L2 cache (64MB), whereas they exceed the typically significantly smaller [33] GPU's L2 cache. As a result, GPU speedup over the CPU drops to 1.8 $\times$ , while it achieves 4.2–5.0 $\times$  speedups when hash tables fit into the GPU cache.

We run the queries with *CPU-GPU co-processing* to evaluate how much we can exploit unified memory. If the GPU outperforms the CPU by a factor of  $x$ , in theory, co-processing can improve over GPU-only execution by at most  $(x + 1)/x$ . In practice, however, this upper bound is reduced by shared memory bandwidth, result-merging overheads, and redundant work. In our implementation, this includes building separate hash tables for CPU and GPU.

Consequently, co-processing is beneficial only when the additional CPU throughput outweighs the cost of redundant work and result merging. For Q11, this holds across all scale factors: the query requires no index structures and produces a scalar sum, making result merging negligible, improving performance by up to 16%.

For Q23 and Q34, co-processing does not improve execution time across all scale factors. At SF1, the cost of building hash tables for both processors and merging results outweighs the benefit of concurrent join probing on CPU and GPU. At SF10, the larger number of probed tuples makes co-processing beneficial on the M4 Pro, where the combined CPU-GPU throughput amortizes this overhead. On the M3 Ultra, however, Q34 does not improve, as the more powerful GPU already executes the query efficiently, making the merge phase comparatively too expensive.

For Q41, co-processing already improves performance at SF1, because the probe phase dominates total execution time more strongly than in the other queries. However, its more expensive merge phase prevents improvements at SF10 on the M3 Ultra, where the GPU's high standalone performance makes the additional CPU throughput insufficient to offset merging costs. At SF50, hash tables exceed the GPU's L2 cache, reducing GPU performance. Therefore, co-processing the probe phase can outweigh the merge overhead and

improve execution time by 44%. Overall, co-processing is most effective when the split work dominates total query execution time. Its benefits diminish when the GPU alone already executes the dominant phase efficiently, or when synchronization and result merging consume a substantial fraction of the saved execution time.

We further observe that data distribution must be carefully balanced, depending on available hardware, query complexity, SIMT thread divergence, and the number of synchronization points. We also noticed occasional delays in kernel dispatch, as CPU threads can become occupied running the CPU-side query.

### 5.3 Discussion

Our results show that GPU execution yields substantial speedups for OLAP workloads on Apple Silicon, and co-processing often provides additional gains.

*Potential Limitations:* Despite the speedups, our prototypes do not mitigate thread divergence [21]. The naïve data distribution could be replaced with fine-grained load balancing [35]. Moreover, our CPU and GPU implementations follow the same structure when executing queries in parallel. Instead, only operators that benefit from GPU execution could be offloaded [14], while reusing intermediate results, e.g., hash tables, to avoid redundant computations.

While we focus on in-memory processing in this paper, workloads *exceeding main memory* require the CPU to handle I/O operations, as the GPU cannot access the disk directly. This raises the question of whether I/O triggers full memory-coherence passes for loaded data (see subsection 3.2). Our experiments indicate that throughput is dominated by disk I/O, and frequent cache flushes effectively synchronize shared memory, avoiding coherence passes.

*Translation to real systems:* Using Apple Silicon's GPU for data processing is non-trivial, as a DBMS must account for its hardware-specific aspects such as unified memory and cache sizes. In particular, the GPU's L2 caches are significantly smaller than the CPU caches, which has to be considered by the query optimizer for offloading decisions at larger data sizes and the choice of cache-local data structures. Synchronization on the GPU is also challenging, as atomic instructions are mostly supported only up to 32 bits. Our proposed spin-lock techniques can be applied to solve this limitation (cf. subsection 4.3), while explicit memory management can shift expensive coherence passes for shared unified memory to DBMS startup (cf. subsection 3.2). Nevertheless, leveraging the GPU or CPU-GPU co-processing in practice remains challenging. Using dynamic co-processing, instead of static data partitioning, requires carefully structured kernels that introduce explicit synchronization points between processors and launch new kernels for morsels, as workloads cannot be synchronized across CPU and GPU using atomic instructions. Finally, a compiling system would require a fast code-generation path that avoids generating Metal text files.

## 6 Related Work

GPU acceleration is a well-established approach for accelerating database systems in both academia [14] and industry [1–4, 16]. Prior work has extensively investigated GPU acceleration of individual database operators, including join [11, 25, 31, 38, 43, 49, 53, 58, 62, 63, 67] selection [54], sort [22, 23, 56], and group-by operations [23, 63], as well as entire queries [26, 50]. While originally proposed for

CUDA, many of these techniques, e.g., parallelization and memory-access optimizations, can also be applied to Metal.

Traditionally, a key limitation of leveraging GPUs was limited memory and comparatively slow data transfer between host and device, which quickly became a dominant performance bottleneck [68]. Prior work has proposed strategies for mitigating this by relying on fast interconnects [37], reducing transferred data volumes through compression [51], and fusing multiple operators into a single GPU kernel [20, 44, 64, 65]. Another approach to reducing data transfer costs is CPU–GPU co-processing, where parts of a query are executed on the CPU while others run on the GPU [48]. However, its effectiveness critically depends on careful load balancing between CPU and GPU [10, 14, 24, 27], as data movement between the two processors often remains the primary bottleneck. We revisit this assumption on Apple Silicon, as unified memory removes explicit transfer overhead and exposes CPU-scale memory capacities to the GPU (up to 512 GB).

Others have looked into using single-chip CPU-GPU systems (APUs) with a unified main memory space that enables substantially tighter cooperation, like inter-operator task placement across CPU and GPU [12], intra-operator work distribution, e.g., for hash joins [27], and CPU-assisted prefetching, decompression, and workload distribution during query execution [28]. APU co-processing has been extended further to heterogeneous execution on edge platforms such as the Nvidia Jetson AGX Xavier [36], and beyond classical APUs, like the Xbox Series X [15]. However, these platforms typically provide less compute than modern consumer-grade hardware and haven't been widely adopted, unlike Apple Silicon.

Until now, Apple Silicon has received little attention in data processing research, likely because its GPU requires programming via Metal. Prior work on leveraging Apple Silicon primarily focuses on large language model training [13, 19, 57, 60], and inference [5, 7–9], often comparing performance against NVIDIA GPUs. Beyond machine learning, Apple Silicon has been compared to NVIDIA GPUs for scientific computing, focusing on single and double-precision performance [32]. More recent work has characterized the CPU and GPU bandwidth and compute performance of low-end M1–M4 devices, reporting up to 85% of the theoretical memory bandwidth [30], aligning with our results.

## 7 Conclusion

We present the first evaluation of Apple Silicon for database workloads. We showed that the unified memory delivers high effective bandwidth for CPU and GPU, and that we can avoid significant overheads for shared areas through explicit memory management, which DBMSs are used to. We further discuss how Metal's programming model differs from other GPU programming models, requiring splitting relational operators into separate kernels at synchronization points and a Metal-specific spin-lock implementation. Still, as our end-to-end evaluation demonstrates, Metal allows accelerating query execution via GPU and CPU–GPU co-processing by up to 5× over CPU-only execution. These findings suggest that leveraging Apple Silicon for data processing is viable and promises significant speedups when its memory and programming model are handled carefully, and synchronization challenges are addressed.

## Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) under the Software Campus program (project REALDRAGON, grant number 01IS23069).

## References

- [1] 2018. *cuDF*. <https://developer.nvidia.com/topics/ai/data-science/cuda-x-data-science-libraries/cudf> (accessed: 03.02.2026).
- [2] 2021. *PG-Storm*. <https://en.heterodb.com> (accessed: 03.02.2026).
- [3] 2023. *BlazingSQL*. <https://blazingsql.com/> (accessed: 03.02.2026).
- [4] 2025. *HeavyDB*. <https://github.com/heavyai/heavydb> (accessed: 03.02.2026).
- [5] Oluwaseun A. Ajayi and Ogundepo Odunayo. 2025. Benchmarking On-Device Machine Learning on Apple Silicon with MLX. *CoRR* abs/2510.18921 (2025). arXiv:2510.18921 doi:10.48550/ARXIV.2510.18921
- [6] Apple Inc. 2025. *Metal Shading Language Specification* (version 4.0 ed.). Apple Inc. <https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf> Accessed: February 13, 2026.
- [7] Wayner Barrios. 2026. Native LLM and MLLM Inference at Scale on Apple Silicon. *arXiv preprint arXiv:2601.19139* (2026).
- [8] Afsara Benazir and Felix Xiaozhu Lin. 2025. Benchmarking and Characterization of Large Language Model Inference on Apple Silicon. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 3 (2025), 48:1–48:26. doi:10.1145/3771563
- [9] Afsara Benazir and Felix Xiaozhu Lin. 2025. Profiling Large Language Model Inference on Apple Silicon: A Quantization Perspective. *CoRR* abs/2508.08531 (2025). arXiv:2508.08531 doi:10.48550/ARXIV.2508.08531
- [10] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209. doi:10.1007/S13222-014-0164-Z
- [11] Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *Proc. ACM Manag. Data* 3, 3 (2025), 165:1–165:26. doi:10.1145/3725406
- [12] Linchuan Chen, Xin Huo, and Gagan Agrawal. 2012. Accelerating MapReduce on a coupled CPU-GPU architecture. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, Jeffrey K. Hollingsworth (Ed.). IEEE/ACM, 25. doi:10.1109/SC.2012.16
- [13] Mu-Chi Chen, Po-Hsuan Huang, Xiangrui Ke, Chia-Heng Tu, Chun Jason Xue, and Shih-Hao Hung. 2025. Towards Building Private LLMs: Exploring Multi-Node Expert Parallelism on Apple Silicon for Mixture-of-Experts Large Language Model. *CoRR* abs/2506.23635 (2025). arXiv:2506.23635 doi:10.48550/ARXIV.2506.23635
- [14] Periklis Chrysogelos, Manos Karpapathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. doi:10.14778/3303753.3303760
- [15] Wei Cui, Qianxi Zhang, Spyros Blanas, Jesús Camacho-Rodríguez, Brandon Haynes, Yinan Li, Ravi Ramamurthy, Peng Cheng, Rathijit Sen, and Matteo Interlandi. 2023. Query Processing on Gaming Consoles. In *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, Norman May and Nesime Tatbul (Eds.). ACM, 86–88. doi:10.1145/3592980.3595313
- [16] Voltron Data. 2021. *Theseus - The SQL Engine for AI Workloads*. <https://voltrondata.com/benchmarks> (accessed: 03.02.2026).
- [17] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1243–1254. doi:10.1145/2463676.2463710
- [18] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33. <http://sites.computer.org/debull/A12mar/hana.pdf>
- [19] Dahua Feng, Zhiming Xu, Rongxiang Wang, and Felix Xiaozhu Lin. 2025. Profiling Apple Silicon Performance for ML Training. *CoRR* abs/2501.14925 (2025). arXiv:2501.14925 doi:10.48550/ARXIV.2501.14925
- [20] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1603–1618. doi:10.1145/3183713.3183734
- [21] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *Proc. VLDB Endow.* 13, 6 (2020), 884–897. doi:10.14778/3380750.3380758

- [22] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPU TeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 325–336. doi:10.1145/1142473.1142511
- [23] Bala Gurumurthy, David Broneske, Martin Schäler, Thilo Pionteck, and Gunter Saake. 2023. Novel insights on atomic synchronization for sort-based group-by on GPUs. *Distributed Parallel Databases* 41, 3 (2023), 387–409. doi:10.1007/S10619-023-07424-2
- [24] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34, 4 (2009), 21:1–21:39. doi:10.1145/1620585.1620588
- [25] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2008. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 511–524. doi:10.1145/1376616.1376670
- [26] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saar, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825. doi:10.14778/3551793.3551833
- [27] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (2013), 889–900. doi:10.14778/2536206.2536216
- [28] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-Cache Query Co-Processing on Coupled CPU-GPU Architectures. *Proc. VLDB Endow.* 8, 4 (2014), 329–340. doi:10.14778/2735496.2735497
- [29] Kijae Hong, Kyoungmin Kim, Young-Koo Lee, Yang-Sae Moon, Sourav S. Bhowmick, and Wook-Shin Han. 2024. Themis: A GPU-accelerated Relational Query Execution Engine. *Proc. VLDB Endow.* 18, 2 (2024), 426–438. doi:10.14778/3705829.3705856
- [30] Paul Hübner, Andong Hu, Ivy Peng, and Stefano Markidis. 2025. Apple vs. Oranges: Evaluating the Apple Silicon M-Series SoCs for HPC Performance and Efficiency. CoRR abs/2502.05317 (2025). arXiv:2502.05317 doi:10.48550/ARXIV.2502.05317
- [31] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, Shimin Chen and Stavros Harizopoulos (Eds.). ACM, 55–62. doi:10.1145/2236584.2236592
- [32] Connor Kenyon and Collin D. Capano. 2022. Apple Silicon Performance in Scientific Computing. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*. IEEE, 1–10. doi:10.1109/HPEC55821.2022.9926315
- [33] Chester Lam. [n. d.]. *A Brief Look at Apple's M2 Pro iGPU*. [https://chipsandcheese.com/p/a-brief-look-at-apples-m2-pro-igpu?utm\\_source=publication-search](https://chipsandcheese.com/p/a-brief-look-at-apples-m2-pro-igpu?utm_source=publication-search) (accessed: 29.04.2026).
- [34] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196. doi:10.1109/ICDE.2018.00026
- [35] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghoshien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peepker, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (2025), 4518–4531. doi:10.14778/3749646.3749710
- [36] Jiesong Liu, Feng Zhang, Hourun Li, Dalin Wang, Weitao Wan, Xiaokun Fang, Jidong Zhai, and Xiaoyong Du. 2022. Exploring Query Processing on CPU-GPU Integrated Edge Device. *IEEE Trans. Parallel Distributed Syst.* 33, 10 (2022), 4057–4070. doi:10.1109/TPDS.2022.3177811
- [37] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1633–1649. doi:10.1145/3318464.3389705
- [38] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1017–1032. doi:10.1145/3514221.3517911
- [39] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. doi:10.14778/2002938.2002940
- [40] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. <https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html>
- [41] NVIDIA Corporation. 2017. *NVIDIA Volta GV100 GPU Architecture: The World's Most Advanced Data Center GPU*. Technical Report. NVIDIA Corporation. <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf> Whitepaper.
- [42] Marius Ottosen, Magnus Keinicke Parlo, and Philippe Bonnet. 2025. DuckDB on xNVMe. CoRR abs/2512.01490 (2025). arXiv:2512.01490 doi:10.48550/ARXIV.2512.01490
- [43] Sungwoo Park, Seyeon Oh, and Min-Soo Kim. 2025. cuMatch: A GPU-based Memory-Efficient Worst-case Optimal Join Processing Method for Subgraph Queries with Complex Patterns. *Proc. ACM Manag. Data* 3, 3 (2025), 143:1–143:28. doi:10.1145/3725398
- [44] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 202–214. doi:10.14778/3425879.3425890
- [45] Andrew Pavlo. 2014. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. Ph. D. Dissertation. Brown University, USA. <https://cs.brown.edu/research/pubs/theses/phd/2014/pavlo.pdf>
- [46] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. doi:10.1145/3299869.3320212
- [47] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2013. Variations of the star schema benchmark to test the effects of data skew on query performance. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*. 361–372. doi:10.1145/2479871.2479927
- [48] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 2 (2023), 11:1–11:38. doi:10.1145/3485126
- [49] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. ACM, 17:1–17:12. doi:10.1145/3085504.3085521
- [50] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1617–1632. doi:10.1145/3318464.3380595
- [51] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1390–1403. doi:10.1145/3514221.3526132
- [52] Louie Sinadjan and Dave Cliff. 2025. GPU-Based Simulation of Evolutionary Spatial Cyclic Games: A Comparative Evaluation of Apple Silicon vs Nvidia. *Procedia Computer Science* 274 (2025), 896–908.
- [53] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 698–709. doi:10.1109/ICDE.2019.00068
- [54] Evangelia A. Sitaridi and Kenneth A. Ross. 2013. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 2013, New York, NY, USA, June 24, 2013*, Ryan Johnson and Alfons Kemper (Eds.). ACM, 4. doi:10.1145/2485278.2485282
- [55] Statista. 2024. *Laptops Worldwide*. [https://www.statista.com/outlook/cmo/consumer-electronics/computing/laptops/worldwide?srsltid=AfmBooodVYz0netOxxE\\_5uky0IGleqOjg2kMf6KdzITISjFcTZeKV6-OH#price](https://www.statista.com/outlook/cmo/consumer-electronics/computing/laptops/worldwide?srsltid=AfmBooodVYz0netOxxE_5uky0IGleqOjg2kMf6KdzITISjFcTZeKV6-OH#price) (accessed: 16.02.2026).
- [56] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 417–432. doi:10.1145/3035918.3064043
- [57] Karol Struniawski, Aleksandra Konopka, and Ryszard Kozera. 2024. Exploring Apple Silicon's Potential from Simulation and Optimization Perspective. In *Computational Science - ICCS 2024 - 24th International Conference, Malaga, Spain, July 2-4, 2024, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 14836)*, Leonardo Franco, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 35–42. doi:10.1007/978-3-031-63775-9\_3
- [58] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. An Empirical Performance Comparison between Matrix Multiplication Join and Hash Join on GPUs. In *39th IEEE International Conference on Data Engineering, ICDE 2023 - Workshops, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 184–190. doi:10.1109/ICDEW58674.2023.

00034

- [59] Gábor Szárnyas. 2024. *DuckDB User Survey Analysis*. [https://duckdb.org/2024/10/04/duckdb-user-survey-analysis?utm\\_source=chatgpt.com](https://duckdb.org/2024/10/04/duckdb-user-survey-analysis?utm_source=chatgpt.com) (accessed: 03.02.2026).
- [60] Tycho FA van der Ouderaa, Mohamed Baioumy, Matt Beton, Seth Howes, Gelu Vrabie, and Alex Cheema. 2025. Towards Large Scale Training on Apple Silicon. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*.
- [61] Vasily Volkov. 2016. *Understanding Latency Hiding on GPUs*. Ph. D. Dissertation. University of California, Berkeley, USA. <https://www.escholarship.org/uc/item/1wb7f3h4>
- [62] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2023. Efficiently Processing Large Relational Joins on GPUs. *CoRR* abs/2312.00720 (2023). arXiv:2312.00720 doi:10.48550/ARXIV.2312.00720
- [63] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1 (2025), 39:1–39:27. doi:10.1145/3709689
- [64] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*. IEEE Computer Society, 107–118. doi:10.1109/MICRO.2012.19
- [65] Haicheng Wu, Gregory F. Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. 44. <https://dl.acm.org/citation.cfm?id=2544166>
- [66] Tianhong Xu, Aidong Adam Ding, and Yunsi Fei. 2025. EXAM: Exploiting Exclusive System-Level Cache in Apple M-Series SoCs for Enhanced Cache Occupancy Attacks. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2025, Hanoi, Vietnam, August 25-29, 2025*. ACM, 1294–1308. doi:10.1145/3708821.3710844
- [67] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational Joins on GPUs: A Closer Look. *IEEE Trans. Parallel Distributed Syst.* 28, 9 (2017), 2663–2673. doi:10.1109/TPDS.2017.2677451
- [68] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. doi:10.14778/2536206.2536210
- [69] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 633–647. <https://www.usenix.org/conference/atc20/presentation/zhang-feng>