Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Accelerating GPU Database Operators on the NVIDIA Hopper Architecture

Semester Project Report

Elvan Konukseven

Supervised by
Prof. Gustavo Alonso and Bowen Wu

Systems Group
Department of Computer Science
ETH Zurich

August 16, 2026

Abstract

Analytical query processing is well suited to GPUs because relational operators expose substantial parallelism and often benefit from high memory bandwidth. Faster CPU–GPU interconnects further strengthen the case for GPU query engines by reducing data-transfer costs. At the same time, NVIDIA Hopper introduces new mechanisms for cooperation and data movement within the GPU, but their suitability for established query-processing algorithms remains unclear. This semester project investigates whether thread-block clusters, distributed shared memory (DSM), and the Tensor Memory Accelerator (TMA) can improve Eiger, a GPU query engine developed by the ETH Systems Group.

We analyze expression evaluation and multiple group-by and join implementations, examining whether their data movement and intermediate state provide useful opportunities for asynchronous transfers, additional on-chip capacity, or cross-block sharing. We find no compelling TMA opportunity in the current implementations: regular input movement is either a small part of execution or provides no reuse, while the dominant custom-kernel work consists of key-dependent lookups, updates, and gathers. We do, however, identify a DSM use case in hash-based group-by. Distributing its aggregation state across a thread-block cluster extends the range in which updates remain on chip instead of falling back to global memory. On an NVIDIA H100 NVL, this design reaches up to 7.4× speedup on uniform workloads and larger gains under strong skew.

We also investigate DSM for partition hash join under skew. When a build partition exceeds one block's hash-table capacity, the baseline divides it into smaller chunks and repeatedly processes the corresponding probe partition. We evaluate two cluster-based designs intended to reduce this repeated probing work, but neither outperforms the baseline. One suffers from remote hash-table accesses and load imbalance, while the other avoids only probe reads that are already served by the cache. Together, these results identify a concrete capacity-driven use case for DSM while showing how remote shared-memory accesses, cluster synchronization, and load imbalance can outweigh the work it removes.

Contents

Abstract . . . . . **i**

1 Introduction **1**

2 Background **3**
2.1 GPU Acceleration for Analytical Queries 3
2.2 Eiger 3
2.3 Hopper Features Considered 3

3 Methodology **5**

4 Operator-Level Analysis **6**
4.1 Expression Evaluation 6
4.2 Grouped Aggregation 6
4.2.1 Hash-Based Group-By 6
4.2.2 Sort-Based Group-By 7
4.2.3 Partition-Based Group-By 7
4.3 Joins 8
4.3.1 Non-Partitioned Hash Join 8
4.3.2 Sort-Merge Join 8
4.3.3 Partition Hash Join and Skew 8

5 DSM Hash-Based Group-By: Design and Evaluation **9**
5.1 Baseline and Capacity Limitation 9
5.2 DSM Design 9
5.3 Experimental Setup 10
5.4 Uniform Workloads 10
5.5 Skewed Workloads 11

6 Partition Hash Join Under Skew **13**
6.1 Skew-Aware Baseline and DSM Opportunity 13
6.2 Distributed Hash Table Under Zipfian Skew 13
6.3 Controlled Partition-Size Skew 15
6.4 Probe Sharing and Matched Results 15
6.5 Why Probe Sharing Does Not Help Here 17
6.6 Lessons from PHJ 18

7 Discussion and Conclusion **19**

Bibliography **20**

Chapter 1

Introduction

Analytical queries scan and transform large columns, form groups, and combine relations. These operations contain substantial data parallelism and are often limited by memory access, making GPUs a natural execution platform [6, 1]. GPU query engines have consequently been studied in both research and industry, but their performance depends on algorithms designed around GPU execution and memory behavior.

Recent hardware and market trends strengthen this motivation. Faster interconnects reduce the cost of moving data between the host and accelerator: Grace Hopper, for example, connects its CPU and GPU through a coherent NVLink-C2C interface with up to 900 GB/s of aggregate bandwidth [3]. At the same time, rapid accelerator development has made older GPU generations increasingly accessible in the cloud at lower prices [9]. Many analytical operators depend more on memory bandwidth and capacity than on the specialized compute throughput used for AI training. Consequently, these GPUs may remain effective for database analytics even after they are no longer state of the art for AI workloads.

These improvements remove some system-level obstacles. Faster CPU–GPU links primarily reduce the cost of reaching the GPU or accessing host-resident data. Once an operator's input is in GPU memory, its runtime is determined by how it reads that data, stores intermediate state, synchronizes threads, and writes results. Exploiting a new GPU architecture therefore requires identifying a concrete bottleneck and adapting the operator so that a hardware feature removes that bottleneck.

This paper applies that principle to Eiger, a GPU query processing engine developed at the ETH Systems Group [8]. Eiger contains several algorithms for central relational operations, including expression evaluation, grouped aggregation, and joins. We study whether thread block clusters, distributed shared memory (DSM), and the Tensor Memory Accelerator (TMA), introduced with NVIDIA Hopper, can improve these existing implementations. We inspect the data flow and working state of each algorithm and determine what cost the feature could replace.

This paper makes three contributions:

1. A systematic applicability study of DSM and TMA across Eiger's expression, group-by, and join implementations, including explanations of where these features are unlikely to help.

2. A DSM-based hash group-by design that extends on-chip aggregation state beyond the shared-memory capacity of one thread block, together with an evaluation of the conditions under which it improves performance.

3. An investigation of two thread-block-cluster designs for partition hash join under skew, using targeted experiments and profiling to explain why neither design outperforms the baseline.

The remainder of the paper introduces GPU query processing, Eiger, and the relevant Hopper mechanisms; describes the operator-level study and DSM designs; and then presents

the experimental evaluation and resulting design guidance.

Chapter 2

Background

2.1 GPU Acceleration for Analytical Queries

A GPU executes large numbers of lightweight threads concurrently. Threads are scheduled in fixed-size groups and organized into thread blocks, which execute on streaming multiprocessors (SMs). This model provides high throughput when many rows can be processed independently and neighboring threads follow similar control flow.

GPU memory is hierarchical. Global memory provides high bandwidth and holds the input columns and large intermediate results, but each access has relatively high latency. Each SM also provides a much smaller shared-memory region that threads in a block can access with low latency. Registers and caches add further levels of locality. Efficient GPU algorithms therefore coalesce unavoidable global-memory accesses, reuse data and intermediate state on chip, and avoid unnecessary synchronization and contention.

Analytical query processing contains many operations that fit this execution model. A scan can assign rows to independent threads; a grouped aggregation can update groups in parallel; and a join can build and probe many hash-table entries concurrently. The challenge is that grouping, joining, sorting, and materialization also introduce irregular accesses, atomics, and intermediate state.

2.2 Eiger

Eiger is a GPU query processing engine whose operators consume and produce columnar data in GPU memory [8]. It provides expression evaluation and several implementations of grouped aggregation and joins, supported by primitives such as sorting, partitioning, reduction, and gather. Query execution can select among these implementations according to properties such as key width, estimated cardinality, and data distribution.

This choice of algorithm is central to the present study because no single implementation performs best for every workload. The alternatives differ mainly in how they organize rows before producing a result. Some access a hash table directly, some reorder rows so that equal keys become adjacent, and others first divide the input into smaller subsets. These choices trade additional data movement for more localized processing and smaller intermediate state. As a result, implementations of the same logical operator can present very different opportunities for Hopper features.

Eiger implements some processing stages with its own CUDA kernels and others using optimized operations from GPU libraries such as CUB, including sorting, partitioning, scans, and reductions. The custom kernels can be modified directly, while changing the internal behavior of a CUB operation would require replacing it.

2.3 Hopper Features Considered

NVIDIA Hopper is the GPU architecture used by the H100 and by the GPU component of the Grace Hopper GH200. In addition to higher compute and memory performance, Hopper introduces GPU-side mechanisms for cooperation between thread blocks and asynchronous

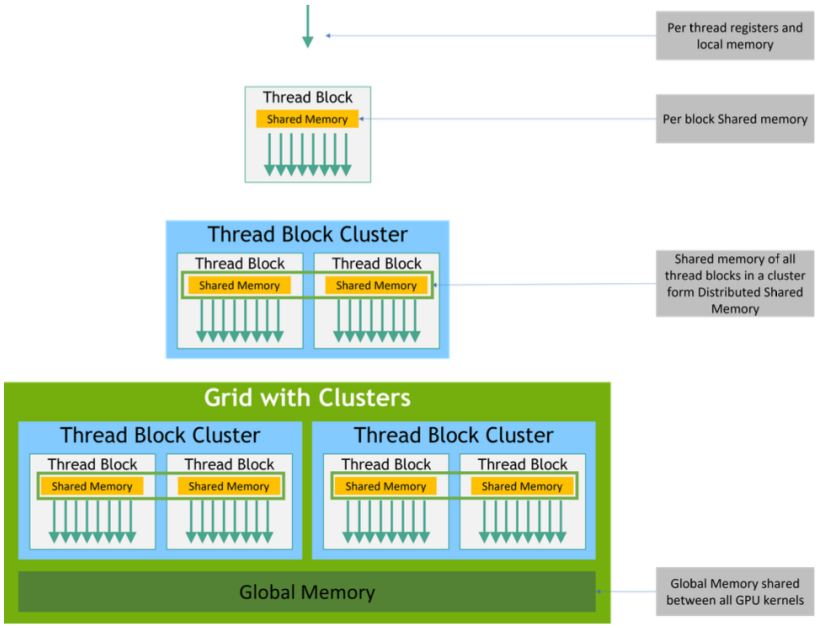


Figure 1: CUDA thread block clusters and distributed shared memory.

data movement. This project considers thread block clusters, distributed shared memory, and the Tensor Memory Accelerator.

Hopper extends the CUDA execution model with thread block clusters. Blocks in a cluster are co-scheduled and can synchronize at cluster scope. Distributed shared memory exposes the shared-memory allocations of those blocks as a cluster-wide address space, allowing a block to access another block's shared memory. With a portable cluster size of up to eight blocks, DSM can expose roughly eight times the shared-memory capacity available to one block, although that memory remains physically distributed and remote accesses require cluster synchronization. Figure 1 illustrates this extension from per-block shared memory to a cluster-wide address space [4].

The Tensor Memory Accelerator is a hardware engine for asynchronous transfers between global and shared memory. A thread can initiate a one- or multidimensional transfer while other threads continue computation, with completion coordinated through barriers. TMA can reduce the instructions required for data movement and overlap transfers with computation, but it requires shared-memory staging. It is therefore most promising when a kernel moves a significant contiguous or predictably strided region that is reused, is costly to address, or can be overlapped with sufficient independent computation to justify buffering and synchronization [2, 4].

Chapter 3

Methodology

For each implementation, we identify its dominant data movement, intermediate state, and synchronization costs, as well as whether Eiger controls the relevant kernels. We then ask whether DSM can provide useful capacity or cross-block sharing, or whether TMA can accelerate a substantial structured transfer through reuse or overlap. In both cases, the expected benefit must justify the additional resource use, setup, and synchronization.

Where the analysis identifies a plausible opportunity, we implement or prototype the corresponding design and evaluate it against the existing implementation. We use controlled benchmarks and profiler measurements where needed to isolate the relevant cost and explain both positive and negative results.

Different algorithms for the same operator organize data and state differently and may therefore expose different Hopper opportunities. A negative result is specific to the implementation and design evaluated; it does not rule out a larger algorithmic redesign.

Chapter 4

Operator-Level Analysis

4.1 Expression Evaluation

Expression evaluation is used inside query operators to compute a formula for each input row. Eiger first converts the requested formula into a sequence of elementary operations, such as loads, arithmetic operations, and comparisons. It can then execute this sequence using either fused or non-fused evaluation. In the fused implementation, each GPU thread reads the input values for one tuple, executes the complete sequence, and writes the final result within a single kernel. In the non-fused implementation, each elementary operation runs as a separate kernel and materializes its result as an intermediate column.

The fused path suggested a TMA pipeline in which the next input tile is prefetched while the current tile is evaluated. It is a plausible candidate because expression evaluation provides independent computation with which a transfer could overlap. Such a pipeline would only be useful, however, if input and output movement represented a meaningful part of runtime.

To isolate that cost, we implemented a streaming control that performs the same input reads and final output write but skips expression evaluation. It takes only about 3–4% of the fused runtime. Even eliminating this cost entirely would therefore provide little overall speedup, while a TMA implementation would additionally require shared-memory buffers and synchronization. We consequently did not pursue this design.

4.2 Grouped Aggregation

A group-by divides the input rows according to one or more key columns and computes aggregates such as sum, count, minimum, or maximum for each distinct key. The number of distinct key combinations, and therefore the number of output groups produced by the operation, is generally not known until the input has been examined, yet it determines both the amount of aggregation state and the level of contention when many rows update the same group.

Eiger provides three families: hash-based group-by assigns dense group identifiers through a hash table; sort-based group-by makes equal keys contiguous and reduces each run; and partition-based group-by performs local hash aggregation after dividing rows by key bits. Their distinct memory patterns create different Hopper opportunities. In the sort- and partition-based implementations, the row-reorganization stages are delegated to CUB, limiting changes to their internal data movement.

4.2.1 Hash-Based Group-By

Hash-based group-by proceeds in two phases. During group discovery, Eiger inserts keys into a global hash table. A global atomic counter assigns each new key a dense group identifier, which later occurrences recover from the table [8]. These identifiers index compact accumulator arrays during aggregation.

In the aggregation phase, blocks process different input rows and any block may encounter any group. If all group accumulators fit in its shared memory, it updates a private copy with shared-memory atomics and later merges one partial value per group into global memory.

The required state is approximately

$$\text{groups} \times \text{aggregates} \times \text{accumulator size}.$$

When this state exceeds one block's capacity, Eiger instead falls back to aggregation directly in global memory. DSM can distribute the arrays across a cluster and delay this expensive fallback, making hash group-by a direct capacity-driven candidate. Chapter 5 evaluates that design.

TMA is less suitable because inputs are consumed once, leaving only limited potential for overlapping their transfer. Staging would also consume shared memory that is more valuable for accumulator state, while the dominant hash-table lookups and scattered updates cannot be transferred in bulk.

4.2.2 Sort-Based Group-By

Hash-based group-by does not reorganize the input, so rows belonging to the same group may be processed by different thread blocks. Each block may consequently need accumulator entries for any of the global groups, and their partial results must later be merged. Sort-based group-by avoids this behavior by ordering the input by group key, making rows with equal keys contiguous. Each run can then be reduced without maintaining a global hash table, which becomes particularly useful at high group cardinalities [8]. Because Eiger delegates sorting and reduction to CUB, modifying their internal data movement would require replacing the underlying primitives. We therefore do not pursue a Hopper-specific modification of this implementation.

4.2.3 Partition-Based Group-By

Partition-based group-by uses the same locality principle without fully sorting the input. It radix-partitions the rows using selected bits of the group key. All rows with the same key are guaranteed to enter the same partition, although different keys may also share that partition. Each partition is then assigned to a thread block, which discovers and aggregates only its local groups using a shared-memory hash table. Because equal keys are brought into the same local work unit, their aggregate can be written to the final output without merging partial results produced throughout the grid.

A partition containing one distinct key can be processed with a block-wide reduction. A partition with a small number of distinct keys uses shared-memory accumulator arrays, while a partition containing too many local groups falls back to global-memory aggregation. Partitioning therefore reduces the accumulator capacity required by each block, although it cannot guarantee that every local group set will fit in shared memory.

This locality is obtained at the cost of a transformation phase. Before aggregation can begin, the operator must construct partition histograms and offsets and reorder the input rows, potentially over multiple radix-partitioning passes. Reducing the number of passes is therefore the most direct opportunity considered here for accelerating the current algorithm.

The current implementation uses 16 radix bits and produces $2^{16} = 65{,}536$ partitions. The CUB implementation analyzed in [8] processes at most eight radix bits per pass on Ampere, so this 16-bit design requires two passes. Reducing the configuration to eight bits would require one pass but produce only $2^8 = 256$ partitions. Each partition would consequently be about 256 times larger on average.

A cluster of eight blocks increases available shared-memory capacity by at most roughly eight times, far below the 256-fold increase in average partition size. DSM therefore cannot make the one-pass design viable by capacity alone and partition-based group-by is therefore not a useful target for a DSM modification.

4.3 Joins

An equi-join combines rows from two input relations when their join keys are equal. Eiger provides non-partitioned hash join, sort-merge join, and partition hash join. These implementations differ in how they organize the inputs before matching: direct hashing builds one global table, sorting places keys in order before merging, and radix partitioning creates smaller co-partitions that can be joined locally. Join execution also includes materialization, where the row identifiers found during matching are used to gather the remaining output columns.

4.3.1 Non-Partitioned Hash Join

The non-partitioned hash join builds a global hash multiset over the keys of one input, probes it with the other input, and records the identifiers of matching rows. It then gathers the requested columns from the original relations to produce the result. Both the hash probes and the subsequent gathers are index-dependent, irregular accesses, leaving no structured transfer for TMA to stage. The implementation uses a global hash multiset shared across the GPU and has no block-local hash state whose capacity DSM could extend. A cluster-local redesign remains possible, but neither feature directly targets the current organization, so we do not pursue them for this implementation.

4.3.2 Sort-Merge Join

Sort-merge join orders both inputs before matching corresponding ranges and materializing the result. Eiger delegates sorting to CUB, while its custom merge and materialization kernels operate on data-dependent ranges and row identifiers. The implementation therefore exposes neither block-local state for DSM to extend nor a structured transfer for TMA to stage.

4.3.3 Partition Hash Join and Skew

Partition hash join first radix-partitions both inputs by the same key bits. Matching keys then fall into corresponding co-partitions, allowing a thread block to build a hash table for a smaller build-side work unit in shared memory and stream the matching probe-side work through it. This replaces random accesses to one large global hash table with faster local accesses, at the cost of the preceding partitioning phase.

Data skew creates a plausible DSM opportunity. When a build co-partition exceeds one block's hash-table capacity, it must be divided into multiple build chunks. Every build chunk is paired with every chunk of the corresponding probe partition. This creates both load imbalance and repeated probe work. A cluster could instead hold a larger build hash table across several blocks, or load a probe tile once and expose it to several block-local hash tables through DSM.

The opportunity is more subtle than increasing shared-memory capacity. A distributed hash table moves remote DSM accesses onto every hash-chain traversal, while probe-data sharing retains most repeated hash lookups and introduces cluster barriers. Chapter 6 evaluates these directions under Zipfian and controlled partition-size skew.

PHJ repeatedly reads the same probe co-partition when an oversized build partition is divided into multiple chunks, initially suggesting an opportunity for asynchronous staging. Within each work unit, however, every probe row is consumed once before a key-dependent hash-table lookup. TMA staging would therefore leave the repeated lookup work unchanged and use shared memory otherwise available to the build hash table. Chapter 6 further shows that the repeated probe reads in the tested skew workload are largely served by L2 rather than DRAM. We therefore do not pursue a TMA variant for PHJ.

Chapter 5

DSM Hash-Based Group-By: Design and Evaluation

5.1 Baseline and Capacity Limitation

The baseline hash group-by has two main stages:

1. Group discovery uses a global hash table to identify distinct keys and assign them dense group IDs. These IDs provide compact indices for the accumulator arrays used in the next stage.

2. Aggregation updates the result columns. If the dense group state fits in shared memory, the block-local path initializes an aggregation buffer in shared memory, performs shared-memory updates, and flushes one aggregate per group to global memory. Otherwise, the global path updates the output columns directly.

The capacity of one block's shared memory therefore determines which aggregation path is used. Once the accumulator arrays no longer fit, the operator replaces on-chip aggregation with more expensive per-row updates in global memory. This change in execution path can cause an abrupt loss of performance as the number of groups or aggregation requests increases.

5.2 DSM Design

The DSM variant on Hopper adds an intermediate path between block-local shared memory and global memory. When the accumulator arrays are too large for one block, their entries are divided across the shared-memory allocations of the blocks in a cluster. Group discovery already assigns consecutive IDs to the discovered groups. These IDs index compact accumulator arrays, whose entries can be conveniently divided across the blocks of a cluster. Threads can then update each block's portion of the accumulator arrays through DSM. After cluster-wide synchronization, each block writes its portion of the completed aggregates to the global output.

This design increases the available on-chip capacity approximately in proportion to the cluster size. It changes only the aggregation stage: group discovery remains unchanged, and the operator retains the original global-memory path whenever the accumulator arrays do not fit across the cluster or cluster execution is unavailable. Correctness was validated across different key and value widths, multiple aggregation functions, and both the DSM and fallback paths.

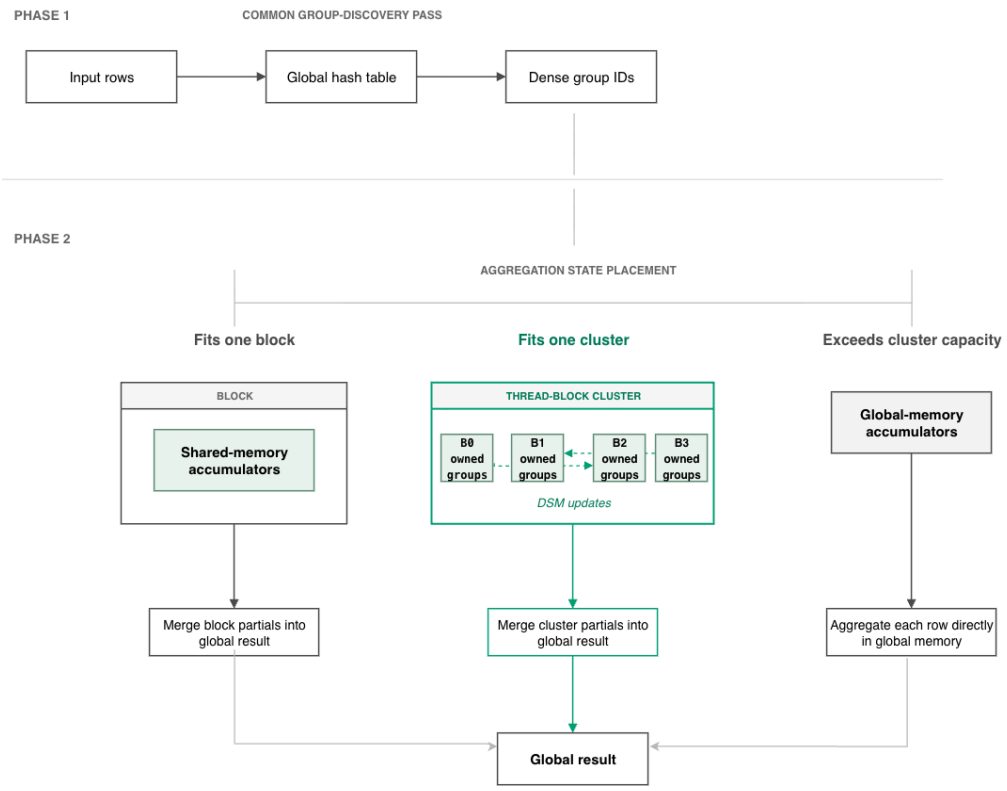


Figure 2: Hash group-by execution paths. Group discovery is common to all configurations. During aggregation, the state is placed in one block's shared memory, distributed across a thread-block cluster, or updated directly in global memory according to its capacity requirement.

5.3 Experimental Setup

The benchmark results were collected on an NVIDIA H100 NVL. All experiments process 2^{29} rows, use 32-bit integer keys and values, and apply Max aggregations. The uniform sweep varies the number of distinct groups and aggregation requests, while the skew sweep fixes two aggregations and varies the Zipf exponent over configured group domains of 16K, 64K, and 256K. DSM is evaluated with cluster sizes 2, 4, and 8. Reported runtimes use the GPU time produced by NVBench from repeated measurements, and speedup compares matching baseline and DSM configurations.

NVBench reported below 5% timing noise for all uniform measurements. In the most skewed baseline configurations, it reached 22%; peak skew speedups are therefore reported approximately.

5.4 Uniform Workloads

Figure 3 confirms the capacity model. When the aggregation state fits in the original block-local shared-memory path, DSM and legacy hash group-by are essentially identical. When it no longer fits in one block but still fits across a cluster, DSM avoids the global aggregation path and becomes faster. Once the state exceeds the selected cluster's capacity, the speedup returns to approximately one because DSM also falls back to global aggregation.

The relevant state size is the number of groups multiplied by the number of aggregations and the accumulator size; group count alone does not determine whether the state fits. With two aggregations, DSM-2, DSM-4, and DSM-8 return to baseline performance at 64K, 128K, and 256K groups, respectively. With eight aggregations, the corresponding transitions occur

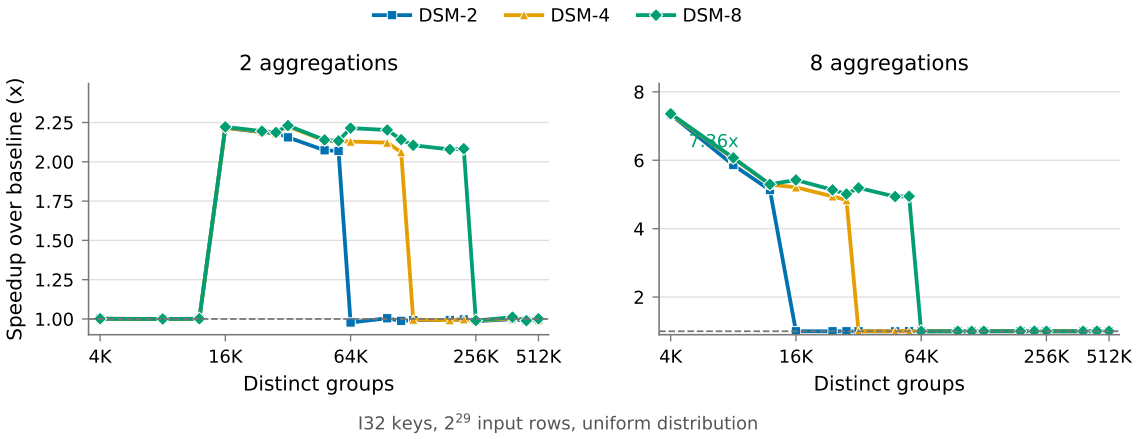


Figure 3: DSM hash group-by speedup as group cardinality increases with two and eight aggregation requests. The three curves represent cluster sizes 2, 4, and 8.

at 16K, 32K, and 64K groups. These abrupt drops are expected capacity transitions, and they show that increasing the cluster size extends the useful DSM range proportionally.

The largest observed speedup in the uniform sweep is about 7.4× at 4096 groups and eight aggregations. At this point, the baseline has already moved to global aggregation while the DSM variants retain the accumulator arrays on chip. Larger clusters do not consistently improve throughput while several sizes fit, but they extend the group cardinality at which this advantage remains available: DSM-2 reaches its capacity first, followed by DSM-4 and DSM-8.

The throughput and aggregation-count sweeps show the same transitions and are omitted here because Figure 3 captures the central result: DSM improves performance only in the interval between block-local capacity and cluster-local capacity.

5.5 Skewed Workloads

Figure 4 evaluates how this behavior changes with a Zipf-distributed key frequency. Skew creates two potential effects. First, repeated updates to a small number of hot groups increase contention in the baseline's global aggregation path. Second, fewer groups from the configured domain may occur in the input, reducing the active aggregation state.

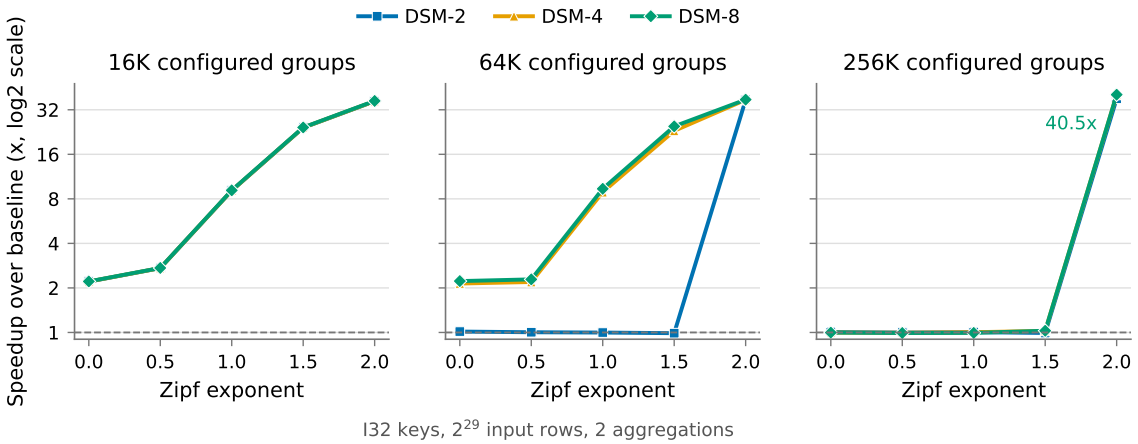


Figure 4: DSM hash group-by speedup under increasing key-frequency skew. The y-axis uses a base-2 logarithmic scale. Each panel fixes the configured group domain and compares cluster sizes 2, 4, and 8.

For 16K configured groups, all three cluster sizes are already in the useful DSM regime under uniform input. Their speedup rises from about 2.2× to about 37× as skew increases, indicating that avoiding contended global updates becomes increasingly valuable. At 64K configured groups, DSM-4 and DSM-8 benefit from moderate skew, while DSM-2 remains near the baseline until Zipf 2.0, consistent with its smaller cluster-local capacity.

At 256K configured groups, all variants remain near the baseline through Zipf 1.5 and then reach approximately 38×–40× at Zipf 2.0. This sharp transition is consistent with a smaller realized group set bringing the aggregation state within cluster capacity, while the stronger concentration also increases contention on the baseline's global updates.

Overall, the results show that DSM can provide particularly large gains under skew when the active aggregation state fits within cluster-wide shared memory while the baseline performs contended global updates. The configured group domain alone does not determine this behavior because strong skew may reduce the number of groups that actually occur.

Chapter 6

Partition Hash Join Under Skew

6.1 Skew-Aware Baseline and DSM Opportunity

Eiger's PHJ uses 16 radix bits, producing $2^{16} = 65{,}536$ co-partitions. One work unit stores at most $B = 4096$ build rows in a block-local hash table and processes at most $P = 8192$ probe rows. We extended the original probe-only work generator to handle oversized build partitions by generating the Cartesian product of build and probe chunks. The number of work units for co-partition i is

$$w_i = \left\lceil \frac{|R_i|}{B} \right\rceil \left\lceil \frac{|S_i|}{P} \right\rceil .$$

Each work unit still builds and probes a hash table in its own block's shared memory, but it must reread and rehash the probe chunk for every build chunk. With 2^{28} build rows and 90% assigned to 64 hot partitions, each hot partition contains about 3.77 million rows and 922 build chunks. Its approximately 4096-row uniform probe co-partition is revisited 922 times.

This repeated work creates two possible uses for an eight-block cluster. First, the blocks could combine their shared-memory capacity into a larger build hash table. In the example above, grouping eight build chunks at a time could reduce the number of times the probe co-partition is hashed and probed from 922 to $\lceil 922/8 \rceil = 116$. Alternatively, the blocks could retain separate local hash tables but load the probe co-partition once per cluster and share it through DSM. This would similarly reduce its reads from the cache and global-memory hierarchy, but each block would still perform the lookup against its own table.

All thread-block-cluster variants evaluated below use their cluster kernels only for build co-partitions larger than the baseline capacity of 4096 rows. Smaller co-partitions continue through the ordinary single-block PHJ path. The reported end-to-end results therefore combine ordinary processing for smaller partitions with cluster processing for oversized partitions.

6.2 Distributed Hash Table Under Zipfian Skew

DSM-HT combines eight blocks into one hash table distributed across their shared memories. Bucket b is owned by block $b \bmod 8$, so build and probe operations may access remote bucket heads and chain nodes. The cluster-wide table can hold up to eight build chunks at once, so a probe chunk is processed once against their combined contents rather than once per build chunk. The larger table therefore reduces both repeated probe-data reads and repeated hashing and lookup against separate build chunks. However, the bucket selected by a probe key may belong to another block, making bucket-head and hash-chain accesses remote DSM operations. Execution has two phases: all blocks first build the distributed table, then they probe it. A cluster-wide barrier is required between these phases to ensure that the table is complete, followed by another barrier before its shared memory can be reused.

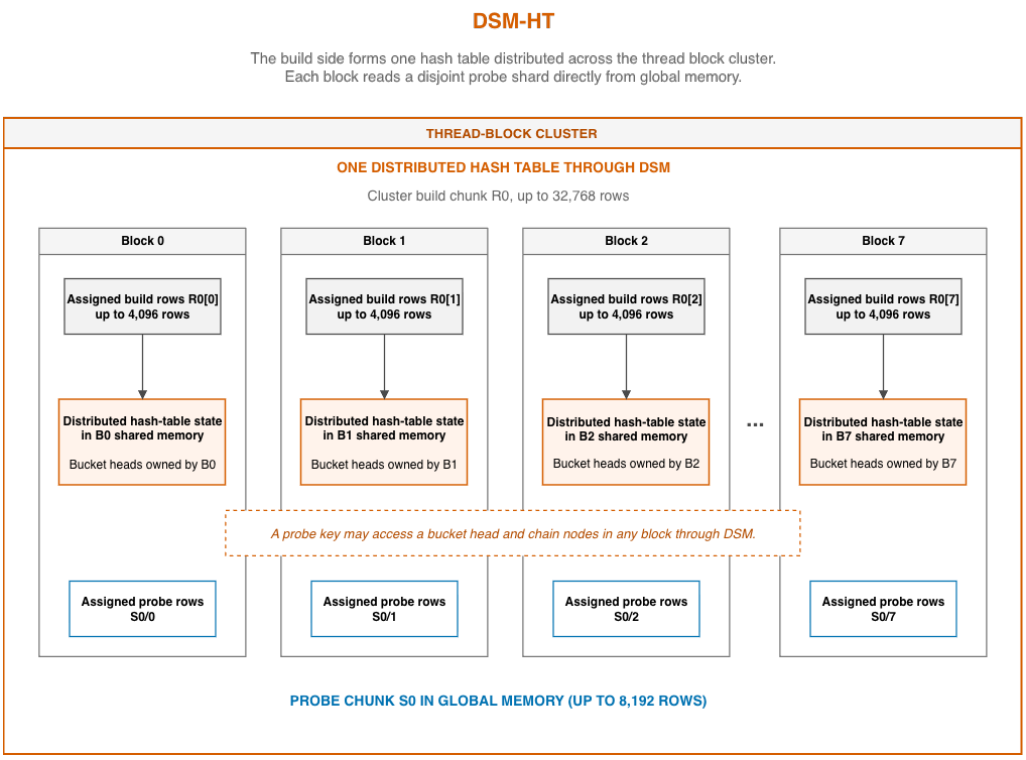


Figure 5: DSM-HT organization. Eight blocks contribute build rows and shared-memory capacity to one cluster-wide hash table; probing may access bucket heads and chain nodes in any block through DSM.

We first generate build keys using a Zipfian distribution. A Zipf factor of 0 produces a uniform distribution, while increasing the factor concentrates more rows on a smaller set of frequently repeated keys. After radix partitioning, this produces both uneven partition sizes and many duplicate build keys within the hot partitions. Figure 6 uses 32-bit keys, four columns per relation, $|R| = 2^{26}$, $|S| = 2^{28}$, and a match ratio of 1.0. PHJ and DSM-HT both take about 27–29 ms for uniform and mildly skewed input. At Zipf 1.0, DSM-HT takes 265.7 ms versus 51.1 ms for PHJ, a 5.2× slowdown; at Zipf 2.0, it is 9.5× slower.

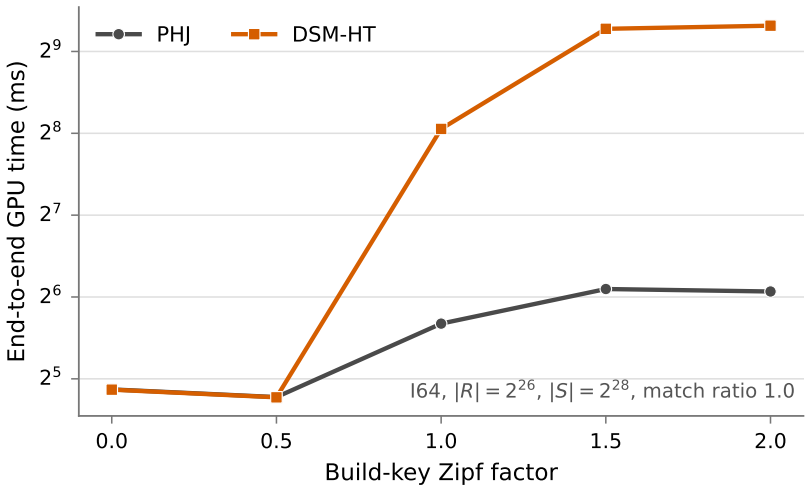


Figure 6: End-to-end PHJ runtime under Zipfian build-key skew. DSM-HT is competitive before duplicate hot keys dominate, then becomes substantially slower. The y-axis uses a base-2 logarithmic scale.

To diagnose this behavior, we use a representative Zipf 1.0 profile with 64-bit keys and otherwise matching relation sizes. Nsight Compute reports 274.4 ms for DSM-HT and 35.9 ms for PHJ. Issue activity falls from 23.0% to 0.91%, while the barrier-stall metric increases by about 44×, showing that the cluster kernel is dominated by waiting rather than useful instruction issue. The implementation explains this behavior: duplicate-key inserts serialize at hot bucket heads, long remotely accessed chains cause blocks to progress unevenly, and cluster barriers hold faster blocks behind the slowest.

Under Zipfian skew, hot keys create both oversized partitions and many duplicate entries in the same hash buckets, making it difficult to separate partition imbalance from hash-table contention. We therefore introduce a controlled workload with unique build keys and deliberately oversized partitions, isolating the repeated probe work caused by partition size.

6.3 Controlled Partition-Size Skew

The Zipf experiment cannot determine whether cluster-wide capacity helps with oversized partitions because duplicate hot keys simultaneously introduce contention and long hash chains. To separate these effects, we generate unique 64-bit build keys and assign them directly to the 2^{16} radix partitions. We place 90% of the build rows in 64 hot partitions and distribute the remaining rows among the other partitions. This produces oversized build partitions without duplicate build keys.

The probe relation contains 2^{28} uniformly distributed rows, or approximately 4096 rows per co-partition. The probe side is therefore not skewed. Nevertheless, for every oversized hot build partition, baseline PHJ rereads the corresponding 4096-row probe co-partition once per build chunk. We use a match ratio of 0.1, meaning that only 10% of probe rows produce a join match. This limits the cost of writing join results, making the repeated hash-table construction and probing work easier to observe. We evaluate build relations of 2^{26} and 2^{28} rows, with four columns in each relation.

6.4 Probe Sharing and Matched Results

For oversized build partitions, **DSM-Probe** keeps the hash tables block-local and uses DSM only to share probe data. In the build phase, each of the eight blocks selects a different build chunk of up to 2048 rows and independently constructs its own hash table. Unlike DSM-HT,

these tables are not combined, so all hash-chain traversal remains in local shared memory. In the probe phase, one probe chunk is divided into eight shards, and each block loads one shard into its shared memory. After a cluster-wide barrier ensures that all shards are available, every block reads all eight shards, one locally and seven through remote DSM, and probes them against its own table. A second barrier ensures that every block has finished before the staged probe data is replaced or released. DSM-Probe therefore avoids remote hash-chain traversal, but each probe row is still looked up once in every local hash table in the cluster.

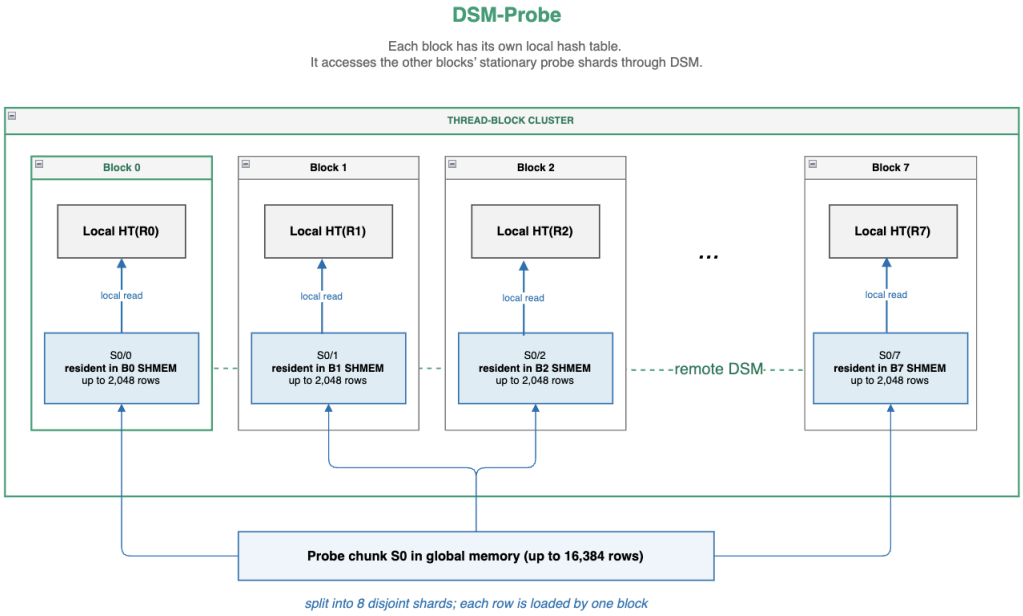


Figure 7: DSM-Probe organization. Each block builds a local hash table and loads one shard of a shared probe chunk. All blocks then process every shard, accessing the other seven through DSM.

To identify where DSM-Probe's overhead originates, we compare it with two intermediate variants. **PHJ-2K** is ordinary PHJ configured with 2048-row build chunks and 2048-bucket hash tables, matching the local build capacity and hash-table size used by DSM-Probe, but without cluster execution or DSM. **Direct-Probe** uses the same local tables, eight-block organization, work units, probe order, and barriers as DSM-Probe, but each block reads the probe rows normally through the cache and global-memory hierarchy instead of sharing them through DSM. PHJ-2K isolates the effect of using the smaller local-table configuration, while Direct-Probe separates the cost of cluster execution from the additional effect of sharing probe data through DSM.

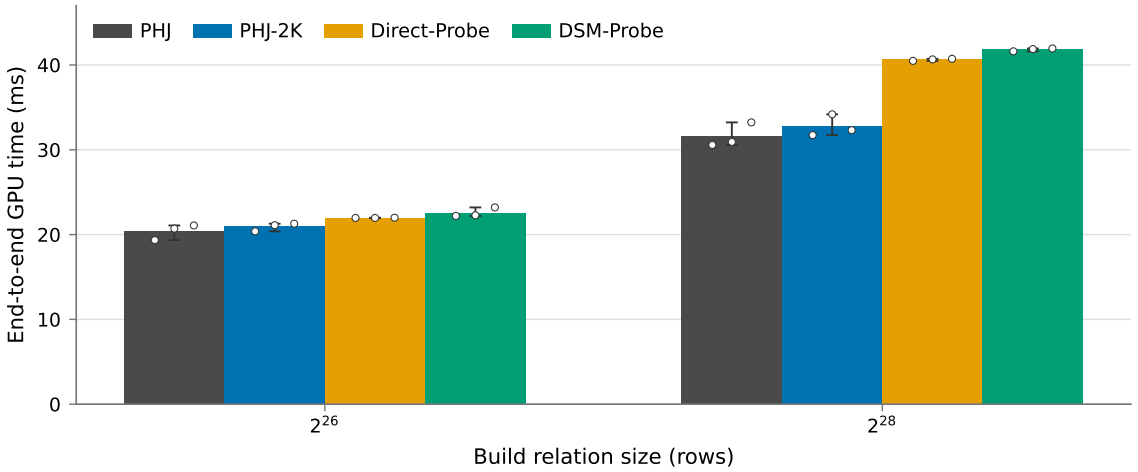


Figure 8: End-to-end runtime of PHJ, PHJ-2K, Direct-Probe, and DSM-Probe under partition-size skew with unique build keys. Bars show the mean across three independent benchmark processes, white points show each process mean, and error bars show their range. The probe relation contains 2^{28} rows, 90% of build rows are assigned to 64 hot partitions, and 10% of probe rows produce a join match.

The comparison separates the costs of the probe-sharing design. PHJ-2K uses the same 2048-row build capacity and 2048-bucket local hash tables as DSM-Probe. It remains within 2.7–3.7% of PHJ, showing that this smaller local-table configuration does not explain the larger slowdown of the cluster variants. Moving the configuration into the eight-block Direct-Probe organization increases runtime by a further 5.0% at 2^{26} build rows and 24.1% at 2^{28}. Replacing normal probe accesses with DSM sharing adds another 2.7–2.9% instead of recovering this overhead. Overall, DSM-Probe is 10.7% and 32.4% slower than PHJ at the two build sizes.

PHJ-2K is not, however, a pure measurement of the effect of repeated probing. Its smaller build chunks generate more probe passes, but they also reduce the hash-table load factor and may shorten hash chains, making each lookup cheaper and potentially compensating for part of the additional probe work. We therefore perform a separate experiment that retains PHJ's original 4096-row build chunks, 2048-bucket hash tables, work units, and launch configuration. The experimental variant repeats all probe reads, hashing, and hash-chain traversal without materializing duplicate matches. Doubling this probe work increases end-to-end runtime by approximately 4% at 2^{26} build rows and 9% at 2^{28}. Repeated probing therefore contributes to runtime, but its cost remains modest relative to the cluster overhead observed at the larger input.

These results also delimit what probe sharing can save. DSM may avoid repeated probe-data loads, but it does not remove the hashing and local-table lookup repeated for every build chunk. Section 6.5 examines whether the avoided loads are sufficiently expensive to justify DSM.

6.5 Why Probe Sharing Does Not Help Here

Table 1 compares Direct-Probe and DSM-Probe at $|R| = 2^{26}$. Both variants execute the same cluster work, use the same local hash tables and barriers, and differ only in how probe data is accessed. Direct-Probe rereads probe rows through the normal cache hierarchy, whereas DSM-Probe stages and shares them through distributed shared memory. Nsight Compute introduces profiling overhead, so we use its results to compare kernel behavior and memory traffic rather than end-to-end runtime.

Table 1: Matched kernel metrics for direct probe reads and DSM probe sharing.

Metric	Direct-Probe	DSM-Probe
Kernel duration	3.660 ms	3.970 ms
DRAM read bytes	970.728 MB	970.809 MB
L2 read sectors (million)	90.93	55.15
L2 read-hit sectors (million)	60.95	25.04
L2 read-miss sectors (million)	30.405	30.402
Executed instructions	1.115 B	1.222 B
Issue active	34.63%	34.97%

DSM-Probe does not help because L2 already captures most of the probe-data reuse. With 32-byte L2 sectors [5], Direct-Probe generates 2.91 GB of L2 requests: 1.95 GB of hits and 0.973 GB of misses. The latter closely matches its 0.971 GB of DRAM reads. DSM-Probe removes approximately 1.15 GB of L2-hit traffic but leaves both L2 misses and DRAM traffic effectively unchanged. It therefore replaces cache-resident reads rather than eliminating expensive off-chip accesses, while adding explicit staging and remote shared-memory operations. The same cache behavior limits the opportunity in PHJ, although PHJ still repeats hashing and lookup for each build chunk.

Published H800 microbenchmarks report 184–213 cycles for remote DSM accesses, compared with 264.5–502 cycles for L2 hits [2]. These measurements use one thread per block following a dependent pointer chain, which prevents accesses from overlapping. DSM-Probe instead issues many accesses concurrently and additionally stages probe shards and synchronizes the cluster. The lower isolated DSM latency therefore does not imply that replacing cache-resident probe reads with DSM will reduce the runtime of this kernel.

In the matched comparison, DSM-Probe executes 106.6 million more instructions and takes 0.310 ms, or 8.5%, longer, while issue activity remains nearly unchanged. Thus, the negative result does not show that DSM has higher intrinsic latency than L2. It shows that replacing L2 hits with DSM does not save enough costly data movement to amortize the additional instructions and staging operations.

Nsight Systems confirms that the slowdown occurs in the matching kernels rather than in the surrounding partitioning and preparation stages. PHJ spends 3.632 ms matching all co-partitions. DSM-Probe spends 2.543 ms processing smaller partitions with the ordinary PHJ path and another 3.972 ms processing oversized partitions with the cluster path, for 6.515 ms in total. Partitioning, scans, transformation, and the remaining GPU work take essentially the same time in both variants.

6.6 Lessons from PHJ

Oversized build partitions create cross-block reuse, but neither evaluated cluster organization converts it into a speedup. DSM-HT combines multiple build chunks into one distributed hash table, reducing repeated probe-data reads and lookups, but places remote DSM accesses on the hash-table traversal path and suffers severe imbalance under Zipfian skew. DSM-Probe keeps hash-table traversal local and shares only the probe data, but retains repeated lookups and replaces cache-resident L2 reads rather than reducing DRAM traffic. Both designs remain slower than PHJ.

These results are specific to the evaluated implementations and workloads and do not rule out other DSM-based join designs. They show, however, that identifying data reuse is not sufficient: the cost removed by DSM must be significant, and exploiting it must not place remote accesses or cluster synchronization on a more expensive execution path. Neither condition holds for the two PHJ designs evaluated here.

Chapter 7

Discussion and Conclusion

The hash group-by results identify a specific DSM use case: the accumulator arrays no longer fit in one block's shared memory but still fit across a cluster. Outside this interval, DSM provides little benefit because the baseline either already aggregates in shared memory or both implementations use global memory. A practical selection policy should therefore use the block-local path when it fits, select a cluster large enough to hold the accumulator arrays when it does not, and otherwise retain the global-memory fallback.

The PHJ experiments show that cross-block data reuse does not by itself make DSM profitable. DSM-HT reduces repeated probe reads and lookups, but introduces remote accesses on the hash-table traversal path and performs poorly under Zipfian skew. DSM-Probe avoids remote hash traversal, but profiling shows that it replaces L2 hits rather than reducing DRAM traffic while retaining repeated lookup work. In both cases, the cost removed by DSM is insufficient to offset the added instructions, staging, synchronization, and cluster execution constraints. The remaining operators likewise expose no comparable capacity-driven opportunity: partition group-by would require a far larger capacity increase than DSM provides, while the central stages of sort-based paths are implemented by external CUB primitives.

We find no compelling TMA opportunity in the current implementations. Expression evaluation provides a plausible overlap pipeline, but equivalent input and output streaming accounts for only about 3–4% of fused runtime. In the hash-based operators, inputs are generally consumed once, so staging creates no reuse and leaves the dominant key-dependent lookups, scattered updates, and gathers unchanged. TMA buffering would also compete with shared-memory hash tables and accumulator arrays whose capacity directly affects the selected execution path.

These conclusions apply to Eiger's current algorithms and the workloads evaluated on the H100 NVL; they do not rule out benefits after a larger algorithmic redesign. Reusable input tiles or substantial transfers that overlap with independent computation could present a different TMA trade-off. Overall, the study contributes both a successful DSM optimization for hash group-by and experimentally grounded guidance for identifying when Hopper-specific mechanisms are unlikely to benefit existing GPU query operators.

Bibliography

[1] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. Revisiting query performance in GPU database systems. arXiv:2302.00734, 2023.

[2] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Hongyuan Liu, Qiang Wang, and Xiaowen Chu. Dissecting the NVIDIA Hopper architecture through microbenchmarking and multiple level analysis, 2025. Preprint.

[3] NVIDIA Corporation. NVIDIA Grace Hopper superchip architecture in-depth. NVIDIA Technical Blog, 2022.

[4] NVIDIA Corporation. NVIDIA Hopper architecture in-depth. NVIDIA Technical Blog, 2022.

[5] NVIDIA Corporation. Nsight Compute Profiling Guide: Memory workload analysis. NVIDIA Documentation, 2026.

[6] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys*, 55(1), 2022.

[7] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *ICDE*, 2019.

[8] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. Efficiently processing joins and grouped aggregations on GPUs. *Proceedings of the ACM on Management of Data*, 3(1), 2025.

[9] Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. Rethinking analytical processing in the GPU era. In *Conference on Innovative Data Systems Research (CIDR)*, 2026.