A Case for Graphics-driven Query Processing

Harish Doraiswamy
Microsoft Research India
harish.doraiswamy@microsoft.com

Vikas Kalagi
Microsoft Research India
t-vkalagi@microsoft.com

Karthik Ramachandra
Microsoft Azure Data India
karam@microsoft.com

Jayant R. Haritsa*
Indian Institute of Science, Bangalore
haritsa@iisc.ac.in

ABSTRACT

Over the past decade, the database research community has directed considerable attention towards harnessing the power of GPUs in query processing engines. The proposed techniques have primarily focused on devising customized low-level mechanisms that utilize the raw hardware parallelism provided abundantly by GPU compute kernels.

In this paper, we advocate a radically different approach – instead of dealing directly with hardware idiosyncrasies, to leverage the well-established *graphics pipeline architecture* baked into the GPU hardware. A variety of advantages accrue from this high-level abstraction: (a) Extracting the power of GPUs is outsourced to highly-optimized graphics drivers, thereby providing hardware-consciousness for free; (b) Query processing becomes agnostic to changes in GPU architectures (e.g. integrated vs discrete) and vendors, requiring only a change of drivers; (c) Contemporary graphics APIs also support a compute element, facilitating query operator designs that seamlessly straddle the compute and graphics worlds.

As a proof of concept of the above vision, we implement here the workhorse Join and GroupBy operators using core graphics primitives. These implementations, based on the Vulkan API, have been evaluated over large benchmark databases on vanilla hybrid computing platforms. The experimental results indicate both substantive performance benefits (typically, around 2X faster) over existing approaches, as well as auto-tuned portability to new hardware platforms.

PVLDB Reference Format:
Harish Doraiswamy, Vikas Kalagi, Karthik Ramachandra, and Jayant R. Haritsa. A Case for Graphics-driven Query Processing. PVLDB, 16(10): 2499 - 2511, 2023.
doi:10.14778/3603581.3603590

1 INTRODUCTION

Over the past decade, GPUs have become mainstream in computing platforms, and it is expected that soon all commodity CPUs will feature an integrated GPU on their real-estate. Given this growing presence, it is no surprise that a considerable body of literature has developed on supporting database query processing over heterogeneous (CPU+GPU) platforms (see [1, 20, 22] for recent surveys).

The proposed techniques primarily use hand-crafted compute kernels and focus on leveraging the abundant raw power inherent in GPU parallelism. Customized low-level workarounds are devised to overcome the: (a) restricted bandwidth between CPU and GPU memories, (b) limited shared-memory capacities of individual GPU nodes, and (c) hierarchical interconnects between GPU elements that lead to non-uniform access times.

These tailor-made solutions have certainly produced impressive performance benefits, reaching up to an order of magnitude [24, 26]. Despite this, GPU-based query processing has not gained traction among industrial-strength database engines, including Microsoft's SQL Server. This is due to two main reasons: First, as highlighted in [20], "system designs have to catch up with the hardware evolution." Second, due to the widespread use of CUDA, the proposed techniques cannot work across hardware from different vendors.

More fundamentally, the original purpose of GPUs – to efficiently support *graphics* operations – has been completely ignored in these formulations. In this paper, we return to first principles, and show how *graphics pipelines*, which are natively optimized to leverage the underlying hardware, *offer a highly attractive platform for database query processing.*

Graphics-based Query Processing

Graphics-based approaches had been contemplated about two decades ago (e.g., [7–9, 29]), but were shelved due to: (a) the graphics technologies of the time being restricted w.r.t. power and programmability, and (b) the emergence of CUDA programming, whose popularity led to raw compute being valorized over graphics components. Notwithstanding, it is our contention that GPU technology has now come of age to the extent that a graphics-driven approach for query processing offers substantively superior capabilities to contemporary GPU-driven techniques.

The graphics pipeline incorporates features to maximize the GPU-based performance of applications such as games. It decomposes the *rendering* operation, which transforms an input 3D scene to a 2D camera image, into a collection of smaller (and simpler) stages, with each stage executed as a collection of parallel threads. This gives the graphics driver the flexibility to schedule threads spanning across stages, thus maximizing GPU utilization.

From a data perspective, the pipeline optimizes writes to images via controlled parallel write operations to *frame buffer objects*, and reads from specific types of GPU memory pages, such as *textures* and *vertex buffers*, through targeted pre-fetching.

*This work was done during a sabbatical year at Microsoft Research India.

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603590

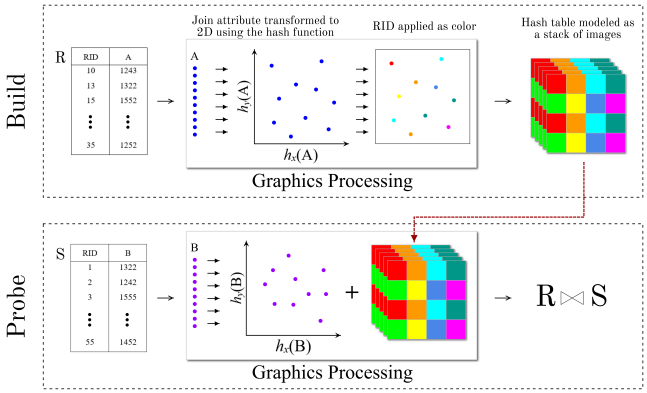


Figure 1: Modeling Hash Join using the Graphics Pipeline

In essence, an application that utilizes the graphics pipeline automatically becomes "hardware conscious" as well. We therefore advocate use of this abstraction for *database query processing*.

Contributions

As a proof of concept of the above vision, we design and implement the workhorse **Join** and **GroupBy** query operators – specifically, their popular hash-based versions – by deconstructing the algorithms into smaller steps amenable to rendering operations that leverage the graphics pipeline features. In this context, as part of the GroupBy operator design, we also propose the first implementation of a GPU-based *multi-attribute* GroupBy.

The toy example of Figure 1 illustrates our Hash Join approach. Here, the join attribute values of relations are represented as a set of 1D points which are transformed to 2D by the hash function. The build relation points are first transformed and rendered using the graphics pipeline to create the hash table, which is represented as a stack of 2D images. The row identifiers are used as color to generate these images. The Join is then accomplished by rendering the probe relation against the hash table.

These algorithms have been implemented using the Vulkan API [15], and evaluated over benchmark databases on vanilla computing platforms. The experimental results indicate substantive performance benefits on large data sets, attaining on average 2X improvements over contemporary customized solutions. We also demonstrate that our approach achieves tuned performance across GPU architectures, essentially for free.

Apart from query processing efficiency, a graphics-based approach also offers several collateral benefits: (a) It is easy to keep pace with technology through new drivers offered by the graphics vendors; (b) Query processing becomes agnostic to changes in GPU architectures (e.g. integrated vs discrete) and vendors, requiring only a change of drivers; (c) Contemporary graphics APIs also support the traditional compute paradigm (similar to CUDA), facilitating query operator designs that seamlessly straddle the best of both (compute and graphics) worlds.

2 BACKGROUND: GRAPHICS PIPELINE

Graphics-intensive applications, such as games, require constant rendering of complex scenes that are continuously changing. The

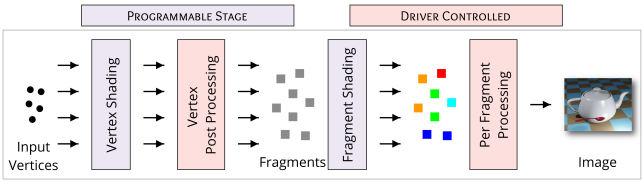


Figure 2: Simplified Graphics Pipeline

rendered scene at a given time step is called a *frame*, which contains geometric objects as seen by a camera at that time step. Achieving smooth transitioning between adjacent frames requires a high frame rendering rate.

GPUs are designed to speed up precisely these operations which are executed as part of the graphics pipeline. The pipeline features a high-level interface (e.g. Direct3D [19] (Microsoft Windows), metal [18] (OSX), OpenGL [27], Vulkan [15]) for easy use of the GPU hardware, while the graphics drivers handle low-level tasks, such as thread scheduling, parallel rasterization, etc. Since these drivers are hardware-specific, applications using the graphics pipeline are automatically optimized for the underlying hardware.

A simplified version of the graphics pipeline (also known as the *shader pipeline*), relevant to our work, is shown in Fig. 2. It is divided into a sequence of stages, some of which are customizable by the developer through the use of programmable *shaders*. In the remainder of this section, we briefly review these pipeline stages – more details are available in [27]. (Our usage here of the pipeline term refers to *intra-operator* constructions, distinct from the inter-operator pipelines associated with query execution plans.)

Vertex Shading. The first stage of the pipeline, the *vertex shading stage*, is typically used to transform the vertices corresponding to the objects in a scene into a common *screen space* (the coordinate system w.r.t. the camera), known as *model-view-projection*. The developer can customize the processing of each vertex through the use of a *Vertex Shader*, which executes in a *Single Program Multiple Data* (SPMD) fashion over the vertices. The input vertices are bound to the pipeline using specialized memory objects called *vertex buffers*. The graphics driver appropriately schedules the threads corresponding to the different vertex shaders, and also prefetches the vertex data from the vertex buffer into the local memory to improve the performance of this stage.

Vertex Post Processing. This stage is handled by the GPU's native driver and is typically used for *clipping* and *rasterization*. *Clipping* is the process where primitives outside the *viewport* (the region visible from the camera) are removed. Primitives such as lines and triangles that partially intersect the viewport are also cropped during this operation, resulting in a new set of primitives that are fully contained within the viewport. Once clipped, the remaining primitives are then *rasterized*, where *rasterization* is the process that converts each primitive into a collection of *fragments*. A *fragment* can be considered as the *modifiable data* corresponding to a pixel. Therefore, the fragment size depends on the screen *resolution*.

Fragment Shading. This is another developer customizable stage that processes all the fragments generated after rasterization, using the logic programmed in a *Fragment Shader*. It is typically used to compute and set the "color" for each fragment. Depending on the required functionality, it can also be used for other purposes

such as discarding fragments, writing to additional output buffers, etc. As with the vertex shading stage, the driver is responsible for scheduling the threads, and appropriately prefetching and setting up the local memory to improve the shader performance.

Post Fragment Processing. The final stage of the pipeline individually processes the fragments that are output from the fragment shading stage, and generates the pixels for the rendered image. Note that there can be multiple fragments corresponding to a given pixel. Such fragments are combined in parallel through an operation called *blending* to finally generate a single pixel.

Virtual Screen. The graphics API also supports rendered images to be output to a "virtual" screen represented by a *frame buffer object* (*FBO*). Each FBO can be associated with one or more *color attachments*. A color attachment is used to represent the color of the pixels, and stores 4 values (r, g, b, a), per pixel, corresponding to the red, blue, green, and alpha color channels. Each attachment is associated with a *texture* that stores the actual color data.

3 EQUI-JOIN OPERATOR

In this section, we present our graphics pipeline-based design of the equi-join operator. In particular, we revisit the classic Hash Join algorithm [3] from a graphics perspective, and deconstruct it into a sequence of rendering passes, thus naturally leveraging the underlying GPU hardware.

Let the join relations be R and S, with $|R| = n, |S| = m$. As with contemporary work [24, 28], we assume for simplicity that the join attributes are 32-bit integers. Thus, the input to the join operator is a set of 32-bit integer pairs (RID, k), where RID is a record identifier in the relation, and k is the value of the join attribute in this row. The extension to 64-bit integers joins is discussed later in Section 7.

We follow the traditional approach of a partitioned hash join, wherein R and S are first partitioned on the CPU into smaller batches that fit in the GPU memory, and these batches are then iteratively processed on the GPU. Further, we choose radix partitioning for the initial step, again similar to [24, 28]. This partitioning is such that the biggest partition can comfortably reside in GPU memory.

Due to the inability to dynamically allocate memory on the GPU to materialize the join, we split the normal two-phase Hash Join into *three* phases, with the incorporation of an additional count phase between the build and probe phases – these phases are described in Sections 3.1 through 3.3. Subsequently, we discuss important design choices in Section 3.4.

3.1 Build Phase

Consider a partition (R_j, S_j), $R_j \subseteq R$, $S_j \subseteq S$ that is being joined. For now, assume that the hash table is built on R_j – the determination of the choice of build and probe relations is discussed later in Section 3.4.3. We represent this hash table as 2D images corresponding to FBOs. Our motivation for doing so is that creating a hash table is akin to rendering images, a common task in graphics applications. Further, probing involves lookups on these images, another common graphics task in the form of texture lookups. Due to their pervasive presence, the GPU hardware is already heavily optimized for performing these tasks.

Within the FBOs, the pixel color channels are used to represent the join attribute values and their RIDs. The pixel locations in the 2D image are determined through hash functions on the attribute values. Specifically, we use a pair of (independent) hash functions, h_x and h_y, to transform attribute value k to a 2D image location:

$$h_x : \mathbb{Z} \rightarrow [0, res_x - 1] \quad h_y : \mathbb{Z} \rightarrow [0, res_y - 1] \tag{1}$$

where (res_x, res_y) is the resolution of each image. Then the hash function h is defined as $h(k) = (h_x(k), h_y(k))$.

This data-to-image transformation is carried out through two core modules of the graphics pipeline – namely, the **Vertex Shader** and the **Fragment Shader**. The Vertex shader maps data elements to FBO locations, while the Fragment Shader blends the data elements assigned to each FBO location.

Our hash-based approach may result in collisions due to two reasons: (a) highly convergent mapping from (domain-bounded) data space to (hardware-limited) image space, and (b) duplicate attribute values. To handle these collisions, we again leverage the graphics modules to create a sequence of *stacked* images: all records that had collisions with the first i images get hashed onto the $i+1^{th}$ image on a first-come-first-served basis. These stacked images are created through iterative renderings of the build table – that is, if the hash table comprises t images, then the build phase requires t rendering passes. Fig. 3 illustrates the first rendering pass of the build phase on a sample input relation R.

Finally, we also maintain a boolean array, **FlagArray**, which is indexed by the RIDs of R_j. It keeps track of the RIDs that have been successfully entered into the stacked sequence of images, and guarantees the correctness of our approach by ensuring that all tuples are eventually processed. The FlagArray is initialized with all its entries being false.

Our algorithms are designed such that, for a given partition pair (R_j, S_j), all of the above structures are fully resident within GPU memory. (In Section 3.3.2, we describe how to handle the skewed case where the data characteristics cause several collisions, and t becomes large enough that the hash stack exceeds GPU memory.)

3.1.1 First Rendering Pass. Before the start of the rendering, the FBO which stores the hash image is cleared so that all its pixels are colored $(0, 0, 0, 0)$. Each record of R_j is then processed as a vertex, implying that R_j itself is bound as a vertex buffer. The Vertex Shader is used to map the input records of R_j to 2D positions assigned by $h(k)$. Recall that each vertex of a vertex buffer is processed in parallel during the rendering. The driver takes care of parallelizing this process as well as prefetching the appropriate vertices into the local memories of the corresponding cores of the GPU.

The transformed vertices are rasterized into a set of fragments by the Vertex Post Processor in the pipeline. These fragments are then processed in parallel by the Fragment Shader. The Fragment Shader for the first rendering pass simply encodes the join attribute-row id pair of the corresponding vertex as the color of that fragment (inset in Fig. 3). Specifically, given the four color channels (r, g, b, a), we set r to be the row identifier, and g to be the join attribute value. We also set the value of b to 1, and this field is used during the probe phase to identify whether or not a pixel of the hash image has a valid value. The a channel remains unused.

All these fragments are then processed by the per-fragment processing stage in the pipeline to write to the corresponding pixels in the FBO, thereby generating the image H_0. When setting up the

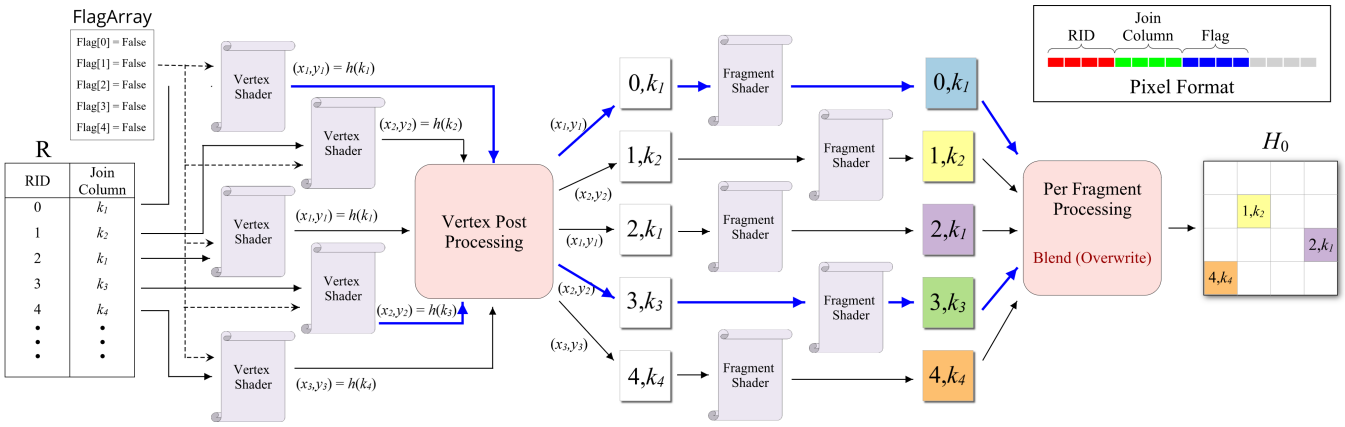


Figure 3: Rendering Pipeline for First iteration of Build Phase. The flows highlighted in blue correspond to tuples that do not succeed in entering H_0 in this iteration. The inset shows the byte-level data layout within a pixel.

rendering pipeline, we set the blend function to *overwrite* – thus, when there is a collision, only *one* of the clashing fragments is (non-deterministically) output to the final image – we refer to this fragment as the *winner fragment*. For example, RIDs 1 and 2 in Fig. 3 generate winner fragments overwriting the fragments generated by RIDs 3 (hash collision) and 0 (duplicate join value), respectively.

After completing the H_0 image construction, we update the Flag Array to reflect the input rows of R entered in this image. This is achieved in a completely parallel operation using a Compute Shader. H_0 is input to this shader as a texture, which processes each pixel to identify if it has been set, and if so, updates the Flag Array to indicate that the corresponding *RID* need not be processed again.

3.1.2 Subsequent Rendering Passes. The subsequent passes operate in the same manner, with the only difference being that in each pass, the hashing scope is restricted to the input RIDs for whom the FlagArray entry is false – that is, those records that have not yet been entered into any of the stacked images. This is accomplished by transforming vertices corresponding to RIDs whose FlagArray entry is true so that they get "clipped" by the pipeline. At the end of each pass, the newly generated fragments are blended together as before to create the hash image H_i corresponding to the current iteration i. This is followed by an update of the FlagArray to reflect the winner fragments in the blending contest. The highlighted flows in Fig. 3 correspond to the fragments that subsequently get hashed in the second iteration to generate H_1 (used as input in Fig. 4).

The build phase on R_j terminates when all entries in FlagArray become true, signifying that all tuples in R_j have been processed.

3.2 Count Phase

Since the size of the materialized join depends on the distribution of the input relations, it poses two challenges: First, the graphics pipeline APIs do not allow dynamic memory allocation *during* pipeline execution, making it necessary to determine the output size *prior* to materializing the join. Second, even if the inputs fit into GPU memory, the join output may be much larger.

To overcome the above challenges, we first compute the size of the materialized join, followed by identifying the location at which to store the output records. This is then used by the Probe phase to materialize the join.

3.2.1 Compute Output Size. This is accomplished using a single rendering pass, as illustrated in the pipeline shown in Fig. 4. Here, as before, the rows in S_j are bound as vertex buffers, and each row is processed as a single vertex in the Vertex Shader. However, unlike in the hash phase, where a single pass rendered all vertices of the input, for the count step, we set the pipeline to be rendered t times, where t is the number of hash images created in the build phase. We employ a feature called *instanced rendering* that is commonly used in graphics applications to efficiently render a specific object multiple times, but each time with different parameters (see Section 3.4).

Thus, for each vertex of S_j, a set of t Vertex Shader instances are spawned. Given a vertex $(RID_S, k) \in S_j$, and $i, 0 \leq i < t$, the corresponding Vertex Shader instance first transforms k to its hash location $h(k)$. It then reads the color (r, g, b, a) of the pixel at $h(k)$ in the hash image H_i. If k is not equal to g, the record pair – r (an RID in R) and RID_S – *do not* form a row in the join output. In this case, the output of the Vertex Shader is simply a location outside the clip region, which therefore does not create any fragment to be processed further. For example, the Vertex Shader processing $RID_S = 42$ and $i = 1$ in Fig. 4 does not create a fragment.

On the other hand, if k is equal to g, then (r, RID_S) forms a valid row in the join output. In this case, the Vertex Shader outputs a location $p(RID_S)$, where p is a function that *uniquely* maps the row in S_j to a location in the output image. This is accomplished as follows: the graphics pipeline provides an implicit vertex indexing that follows the order of the vertices in a vertex buffer. This is provided as an in-built variable in the Vertex Shader, and thus has a one-to-one mapping between RID_S and the vertex index v_{in}. The function p is then defined as

$$p(RID_S) = p^{1\rightarrow 2}(v_{in}(RID_S)) \tag{2}$$

where $p^{1\rightarrow 2}$ maps an integer to a 2D coordinate using a row-major ordering.

The Fragment Shader, when processing this fragment, simply sets its color to $(1, *, *, *)$ – here $*$ denotes a don't-care value indicating that the color channel is not used. The blend function for

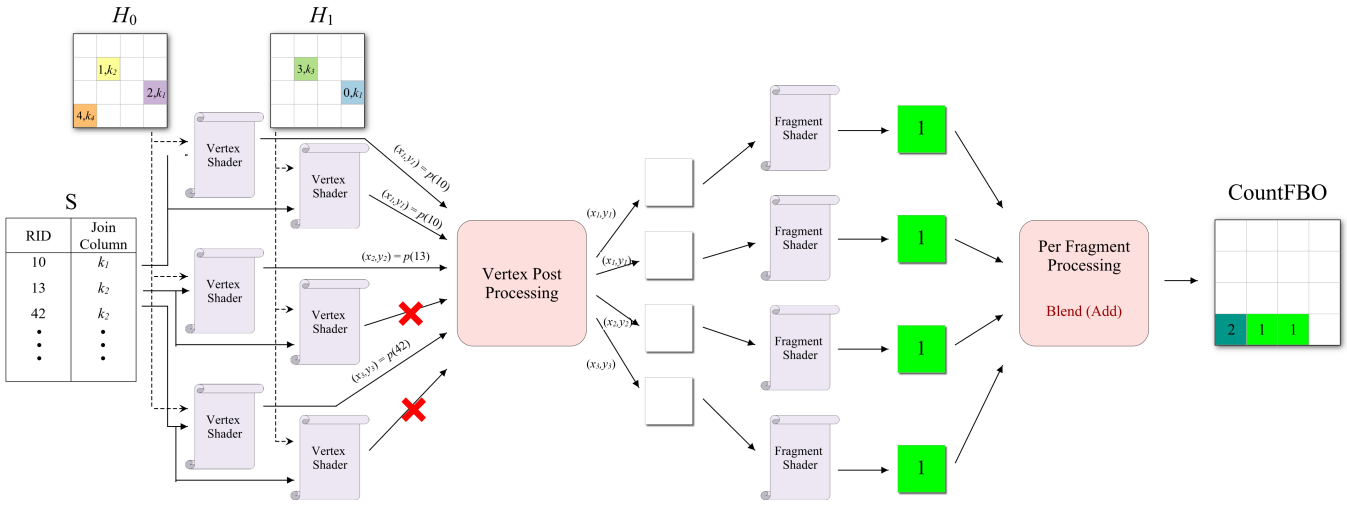


Figure 4: Rendering Pipeline for Count Phase

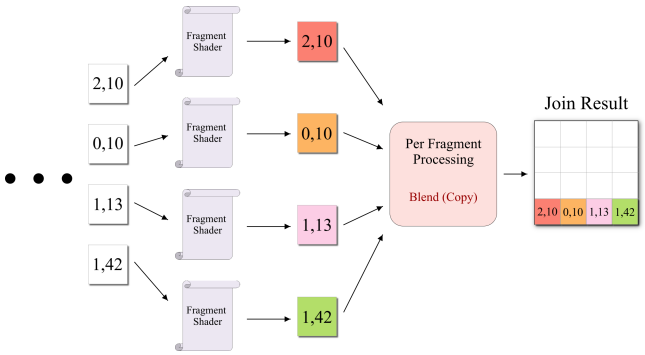


Figure 5: Rendering Pipeline for Probe Phase

this pipeline is set to *add* the colors (note the difference from the Build Phase, where the blend function was *overwrite*). This means that the Per Fragment processing stage simply adds all the colors of the fragments corresponding to a single pixel, and sets the color of the pixel to this sum – semantically, it captures the number of rows in R_j that match a given row in S_j. Thus, at the end of the rendering pass, each pixel of the FBO stores the number of output rows corresponding to the input row associated with that pixel location. We call this the *CountFBO*.

3.2.2 Output Location Identification. Apart from the size computation, we overload the Count step to also identify in advance the *location* in the output array where each result row should be stored. Specifically, we perform an exclusive prefix sum of the elements of *CountFBO* and store it as *CountArr*.

Assume that $|S_j|$ is less than the size of *CountArr* (Section 5 describes how we ensure this inequality). After this operation, the value of the last element of *CountArr* captures the cardinality of the join output. Further, if we were to construct a linear array of this cardinality, the output corresponding to record RID_S would be stored in the array from index $CountArr[p(RID_S)]$ onward.

3.3 Probe Phase

The main idea during the probe phase is to test and match each record in S_j with the t pixels (one per hash image) corresponding to the location defined by the hash function $h()$. Next, we first describe the pipeline used for the probe phase, followed by discussing how a large hash stack size (i.e. large t) is handled.

3.3.1 Probe Pipeline. The pipeline used for the probe step is similar to that used for the count step. In addition to the hash images, the Vertex Shader also takes as input *CountArr* that was previously computed. If a vertex $(RID_S, k) \in S_j$ matches with the image H_i to produce an output record, the location of this output is computed using the *CountArr* as:

$$loc = \text{atomicAdd}(CountArr[p(RID_S)], 1)$$

where *atomicAdd()* atomically increments the value at *CountArr[p(id)]* by 1, and then returns its original value prior to the increment.

The above strategy provides a lock-free mechanism for identifying the index within the output array. The actual 2D location where the join output is written to in the FBO is then obtained using the function $p^{1\to 2}(loc)$, where $p^{1\to 2}$ is the same 1D-to-2D function as that used in Equation 2. The Vertex Shader also passes along the values of the matched row ids, i.e., RID_S and $RID_R = r$ (recall that r was obtained using a texture lookup on H_i).

The rasterized fragments are then simply written to the result FBO at their designated locations by the Fragment Shader. The colors of these fragments are set as $(RID_R, RID_S, *, *)$, denoting the matching join records. This part of the pipeline is illustrated in Fig. 5. At the end of the rendering pass, the result FBO contains the materialized join identifiers.

Note that it is possible that the physical size of the FBO, used for materializing the join, turns out to be smaller than the join output size computed in the previous step. In this case, the probe step is split into multiple rendering passes such that the output size from each pass can be accommodated within the FBO. The specific records in S_j that are processed in each pass are determined using

the *CountArr* – it is used to partition S_j such that the join resulting from each partition fits within the FBO. At the end of each pass, the result FBO is transferred to the CPU before starting the next pass.

3.3.2 Hash Stack Height Threshold. Given the limited memory available in the GPU, it will not be able to handle a large hash stack size on the R_j partition. Therefore, we limit the stack height of the hash images to a fixed number, *SH_limit*, that can comfortably fit in GPU memory. Consider the case where the stack height required for the join is found to be greater than *SH_limit*, arising out of a large number of duplicates or hash collisions. In this case, we pause the hash computation after *SH_limit* images are generated, then run the probe step, and subsequently generate the next stack of hash images. Note that this punctuated process only requires that the flag array F be kept current, and feeding F to the rendering pipeline when the next hashing cycle is initiated.

3.4 Design Choices

As described above, our join algorithm makes several design choices to leverage the rich features of the graphics pipeline. The obvious advantage of this approach is that hardware tuning is automatically obtained by simply being pipeline conscious. Specifically, we derive the following benefits: First, by processing the data similar to vertices, the driver manages not only the parallelism, but also the necessary data allocation across different thread groups to optimize fast vertex data reads (including data pre-fetching). Second, as mentioned earlier, by making use of images to store the hash tables, we take advantage of not only faster "texture lookup" operations, but also more efficient data writes into the FBOs.

Additionally, we have also made the following choices to maximally harness the graphics features, as well as to overcome some of the graphics-induced limitations.

3.4.1 FBOs vs Traditional Buffers. To materialize the join, it is possible to use traditional buffers (called storage buffers) instead of writing the output to an FBO (like in our approach). However, unlike write to storage buffers which are immediate, writing to an FBO is lazy, wherein the actual writes occur post the Fragment Shader stage. Since the writes are driver controlled, thus allowing driver-based scheduling of the write operations. Furthermore, it allows the usage of the in-built blend operations.

3.4.2 Instanced Rendering. A straightforward way to design the count and probe pipeline in the probe step would have been to use a Vertex Shader similar to that of the hash phase. That is, map each row of relation S into its hash location, and then use the Fragment Shader to lookup the stacked hash images and test for equality between the join attributes. This process involves looking up t textures in a loop within the Fragment Shader, resulting in multiple random memory accesses. Further, it can generate multiple output records due to which efficient FBO writes cannot be used (a fragment shader can write to only one location of the FBO).

Therefore, we instead chose to use Instanced Rendering since it provides the following benefits: First, since each Vertex Shader and Fragment Shader instance has constant time complexity (because of no loops), the graphics driver can now efficiently schedule the parallel threads to maximize GPU utilization; Second, each Vertex Shader now reads from exactly one texture, which gives the driver leeway to appropriately schedule threads and the corresponding texture prefetches. Third, we get to use the efficient FBO writes to output the join results.

3.4.3 Minimizing the Hash Stack size. Consider a join being performed on a partition (R_j, S_j). As the number of hash images increases, the number of rendering passes (in the build phase) and the number of vertices rendered (in the probe phase) also increase. This will in turn slow down the performance of the join. Thus, we ideally want to build a hash stack that has fewer images.

To accomplish the above goal, we add a *compute_statistics* step before the build phase, which counts the number of hash images required for R_j and S_j, respectively. The hash images are then dynamically built on the partition (R_j or S_j) with the smaller number of images, resulting in what may be termed as a **"SnakeJoin"** where the build can switch between R and S partitions and is not exclusive to a single relation.

The compute statistics step is also executed using the graphics pipeline as follows. It uses the same Vertex Shader as the hash phase. However, in the Fragment Shader, it simply outputs the color $(1, *, *, *)$ and sets the blend function to add the fragments. Thus, at the end of this rendering, each pixel of the output image will have the number of clashes corresponding to that pixel location. The maximum value over all these pixels gives the number of hash images that are required.

Since the computation is done in a single rendering pass, its overhead is small, especially when compared to the potential degradation due to building hash images on the wrong partition.

3.4.4 Handling Large Hash Stack Sizes. In case of skewed data, it is possible that the hash stack size may become infeasibly large from a memory perspective. This, despite the sparsity of entries in the individual images since the memory is pre-allocated. To handle such pathological cases, we introduce an additional pass during the build phase to flatten the hash stack into a single dense image. This is achieved as follows: Since the compute statistics step described above also counts the number of collisions for each pixel of a hash image, we make use of this information to map each location to contiguous pixels in a single image, thus reducing the number of images and the memory occupancy.

The flatten image step is introduced during runtime depending on the hash stack size, and this information is then used during the probe phase as well.

4 GROUPBY OPERATOR

In this section, we describe the algorithm for the **GroupBy** operation, which poses some unique challenges as compared to Join processing. We begin in Section 4.1 with the graphics-based approach for a single-attribute grouping – that is, where the input comprises n pairs of the form (k, val), where k is the 32-bit integer grouping attribute, and val is the associated value that feeds into an aggregate operation. Subsequently, we follow up in Section 4.2 with the extension to multiple GroupBy attributes, which requires additional graphics machinery.

Similar to the Join approach, we handle data that does not fit in GPU memory by first partitioning the input table into batches that can be comfortably GPU-resident, and then iteratively processing

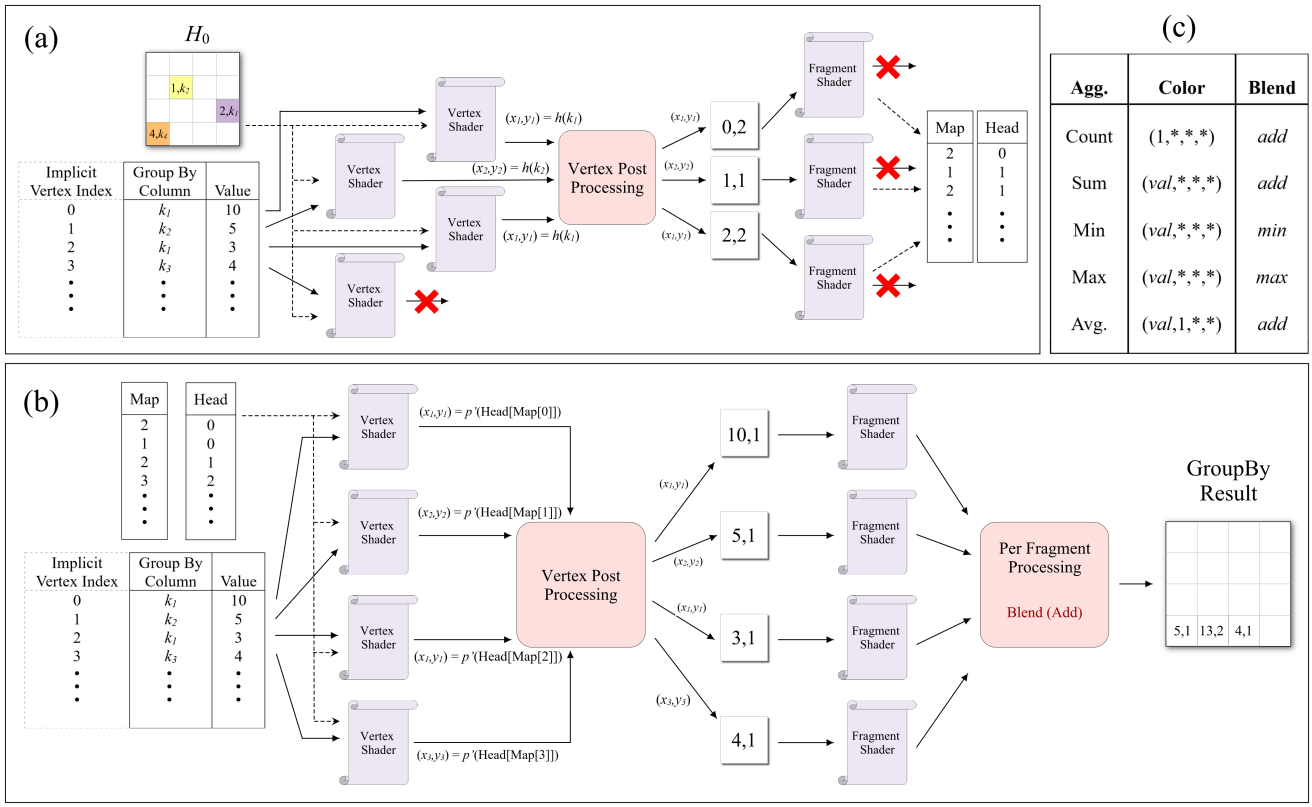


Figure 6: GroupBy algorithm. (a) Second Rendering Pipeline in Grouping Step. (b) Rendering Pipeline for Computing Aggregates (here, average). (c) Colors and Blends for Aggregates.

these batches. We use the same radix partitioning-based scheme as before to identify the batches. Here, an interesting issue that arises in the multi-attribute GroupBy context, is the specific choice of partitioning attribute. We make the choice based on statistics computed during query execution, as described in Section 4.2.2.

Our approach can handle distributive and algebraic aggregate functions [10]. The current implementation specifically supports *count*, *sum*, *minimum*, *maximum*, and *average* functions.

4.1 Single Attribute GroupBy

4.1.1 Grouping Step. The initial grouping step uses iterative processing of two rendering pipelines. The first is similar to the hashing pipeline used in join, creating a series of hash images on the grouping attribute k. However, a critical difference is that a new image is created only for hash collisions, but not for duplicate attribute values. The implicit vertex index provided by the graphics pipeline is used as the RID while generating this image.

The hash image created in each iteration is used by the second rendering pipeline to map each tuple of the input table to a unique position defined by the grouping value. This pipeline is illustrated in Fig. 6(a). The vertex shader maps the RID v_{in} (implicit vertex index) to the RID of the hashed vertex obtained by looking up $H_i[h(k)]$, where k is the GroupBy attribute of the vertex being processed, and i is the iteration number. If they are equal, then the RID v_{in} is mapped to the RID in $H_i[h(k)]$. In essence, this mapping is similar to the *union-find* data structure: when multiple tuples share the same attribute value, one of them is identified as the head for the group, and all the others point to this head. For example, in Fig. 6(a), RIDs 0 and 2 corresponding to group-by attribute k_1 get mapped to 2. So, the head for this group is Record 2. This mapping is stored using a buffer called *Map*. The records corresponding to the heads of the different groups are stored in a separate buffer called *Head*. In particular, *Head* stores a 1 for records that form the heads of their respective groups, and 0 for the others.

The pair of rendering pipelines are run iteratively until no more collisions are present, and all the input records have been processed. Note that since the hash image generated in an iteration is immediately consumed, it is not necessary to retain multiple images.

4.1.2 Aggregation Step. Before computing the aggregation, a prefix sum operation is first executed on the *Head* array. It is precisely to facilitate this operation that we write directly to storage buffers instead of an FBO in the grouping step. Had an FBO been used, the associated *Map* and *Head* values for a record would be stored within a single pixel of the FBO, making the prefix-sum computation cumbersome. The prefix-summed *Head* provides the required output location index for each group.

The aggregation is then accomplished in a single rendering pass, as illustrated in Fig. 6(b). Here, the Vertex Shader uses the *Map* and *Head* buffers to transform a given record to its mapped location.

Specifically, when processing a record with vertex index v_{in}, its group is given by $Map[v_{in}]$, and its location in a 1D array is provided by $loc = Head[Map[v_{in}]]$. The location in the output FBO is then obtained by using the mapping $p^{1\rightarrow 2}(loc)$ (Equation 2).

The fragment shader then simply outputs the appropriate color depending on the aggregate function. These colors are "blended" together in the per-fragment processing stage to generate the final output FBO, which is then returned to the CPU. The different blend functions and the colors used for the different aggregate functions are as shown in Fig. 6(c). For the case of *Avg*, there is a post-processing step which computes the average by dividing the r channel (which stores the sum of the output pixels), with the g channel (which stores the count of these pixels). The Compute Shader is invoked for this evaluation.

4.2 Multi-Attribute Group By

We now move on to considering Multi-attribute GroupBy, where the query features l GroupBy attributes $k_1, k_2, \ldots, k_l$. As with the single attribute GroupBy, the **Grouping** step hashes only one of the attributes. In Section 4.2.1, we first discuss our algorithm which assumes k_1 to be the hashing attribute. Then, Section 4.2.2 describes how this attribute is chosen from the l candidates.

4.2.1 Processing Multiple Attributes. Different from the single attribute GroupBy, the Fragment Shader writes to the hash FBO *all* the attributes $k_1, k_2, \ldots, k_l$. This raises a storage issue that was not present either for Join or single-attribute GroupBy. Specifically, an image pixel can store only 4 values corresponding to r, g, b and a, of which 3 are already used to store the RID (r), GroupBy attribute (g), and validity flag (b). Therefore, at most, one more grouping attribute can be natively accommodated in the pixel. But, if $l > 2$, then we have to bring in additional graphics machinery to handle the overflow – our solution is to use multiple *color attachments* (see Section 2) for the FBO.

In particular, given a query with l grouping attributes, a total of $\lceil \frac{l+2}{4} \rceil$ attachments are required to accommodate them. Thus, attributes $k_i, i \geq 2$ onward are written to the $\lceil \frac{i+2}{4} \rceil^{th}$ attachment of the FBO. The mapping pipeline is modified to compare with all of these attributes to identify a clash during the mapping phase of the grouping step. This is implemented by binding all the attachments as *textures* in the second grouping pipeline.

The remaining steps of the multi-attribute algorithm are the same as those of the single-attribute GroupBy.

4.2.2 Choosing Attribute Order. For multi-attribute GroupBys, we would like to choose the attribute that is most diverse for the hashing operation. This is because a less diverse attribute can cause more collisions when coupled with the other attributes. Thus, if the database already stores statistics about the GroupBy attributes, then the one with the most number of unique values is chosen to perform the hash. In the absence of such statistics, we piggyback on the partitioning step to identify the hash attribute. Specifically, we compute fine-grained histograms on all the GroupBy attributes as part of the first step of the radix partitioning, and use the number of non-empty bins as a heuristic to approximate the diversity of the attributes. The partitioning is then performed using the chosen attribute, which continues to be used in the grouping step.

5 IMPLEMENTATION

The Join and GroupBy operators were implemented in C++ and use the recently developed feature-rich Vulkan API [15] to access the graphics pipeline. There are several challenges that have to be overcome when using the GPU, and in particular the graphics pipeline. First, the use of images imposes restrictions on the maximum resolution of the image, as well as the amount of information that can be stored per pixel of the image. Second, the lack of a common memory space between the CPU and GPU, along with the limited GPU memory, can make data transfer overheads a bottleneck. Third, dynamic memory allocations are not possible during the execution of the pipeline.

Some of the above challenges were handled in the design of the algorithms in the previous sections (e.g., multiple attachments, additional count step). In this section, we focus on the choices made in our implementation to handle the other challenges.

Setting Image and Partition Sizes. A core parameter of our algorithms is the resolution (or size) of the hash images. While in principle, more would appear to be better, an excessively large resolution can be detrimental by slowing down the rendering. Due to this, graphics drivers set a limit on the maximum image sizes supported by the hardware. Accordingly, we chose a resolution of 4096×4096, a reasonable setting for contemporary GPUs, which allows hash images to host up to 16 million entries.

Given this hash image size setting, the data partition sizes need to be assigned so as to limit the number of collisions to an acceptable level. Specifically, the expected percentage of collisions [2] is given by $50 * \frac{PartitionSize}{HashImageSize}$ – if we set this level to a modest 20 percent, then the Partition Size works out to around 6 million entries for the 16 million image size.

Image Storage Efficiency. Improving the space usage of a hash image not only reduces the space utilization, but also increases the number of images that can be stored in the GPU. However, our pixel layout thus far, with its four 32-bit color components, is highly wasteful – for instance, b stores only a boolean flag, and worse, a is completely unused. To improve storage efficiency, we take recourse to: (a) Replacing the 32-bit color channels with 16-bit color channels, (b) Spreading values across multiple channels as shown in Fig. 7,

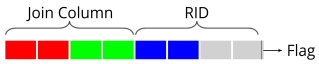


Figure 7: Data layout for 16-bit color channels

and (c) Replacing RIDs with the 31-bit implicit vertex indices (since the number of rows processed per data *partition* is less than 2^{31} by design) provided by the vertex shader and using the remaining 1 bit to store the flag. These simple modifications result in *halving* the space requirement of a hash image.

Hash Stack Height Threshold. As discussed in Section 3.3.2, the stack height of the hash images is limited to *SH_limit*. In our implementation, SH_limit is set to 8, corresponding to 1GB of GPU memory, easily manageable even in lower-end GPUs.

Parallel Rendering. The partition sizes defined above are dependent on the rendering capability of the GPU hardware. However, performing just one rendering at a time may not completely utilize the available GPU resources. Thus, to maximize the GPU utilization, we perform *parallel* renderings, i.e., process partitioned batches in

parallel to the extent that the memory is fully utilized. A crucial advantage of this strategy is that our implementation can easily scale to architectures featuring multiple GPUs.

Pipelining Compute and Data Transfer. To improve GPU efficiency and reduce data transfer bottleneck, our implementation also pipelines data transfer with compute. Specifically, in each thread associated with a single rendering, the data transfer of a batch $i+1$ to the GPU is performed in parallel with the compute corresponding to batch i of that thread.

6 EXPERIMENTAL EVALUATION

In this section, we first describe our experimental setup, and then present the performance profiles for the Join and GroupBy implementations. The performance metric is the end-to-end execution times of the operator, that is, from the time the input relation(s) are loaded into CPU memory, to the time the entire output is materialized in CPU memory.

6.1 Setup

Our evaluation was conducted on a Windows 10 desktop provisioned with AMD Ryzen 5950X CPU, 128GB memory, and Nvidia RTX 3080 LHR GPU with 10GB VRAM and a PCIe interface. The source code of the database operators was compiled using Visual C++ 2019, with the relevant optimization and AVX flags.

6.1.1 Data sets and queries. We report the results primarily from Joins and GroupBys that were run on a **1 Terabyte** version of the TPCH benchmark database, thereby simulating industrial-strength data warehouses. We use the following four Joins on this database in our evaluation:

JO1. Customers $\bowtie_{custkey}$ Orders
JO2. Part $\bowtie_{partkey}$ Lineitem
JO3. Partsupp $\bowtie_{partkey}$ Lineitem
JO4. Orders $\bowtie_{orderkey}$ Lineitem

We also evaluated the join performance on the "narrow-schema" databases used in previous work [24, 28]. In particular, we use both uniformly generated data sets as well as skewed data sets.

To evaluate single attribute GroupBy performance, we run the following GroupBy operations:

SGO1. $_{orderdate}\mathcal{G}_{SUM(extendedprice)}$LineOrder
SGO2. $_{orderkey}\mathcal{G}_{SUM(extendedprice)}$LineOrder
SGO3. $_{partkey}\mathcal{G}_{SUM(supplycost)}$PartSupp

The queries used for multi-attribute GroupBy are:

MGO1. $_{orderkey,orderdate}\mathcal{G}_{SUM(extendedprice)}$LineOrder
MGO2. $_{orderkey,orderdate,shippriority}\mathcal{G}_{SUM(extendedprice)}$LineOrder

Here, **'LineOrder'** is the materialized join between Lineitem and Orders in the TPCH database, and the LineOrder queries are designed to simulate a simplified version of TPCH Query 3.

6.1.2 Baselines. Our goal is to assess the performance benefits of incorporating the graphics pipeline to implement query operators, as compared to the traditional wisdom of using compute APIs such as CUDA or OpenCL. We refer to our driver-based approach as **GARUDA** (Graphics ARchitecture Utilized for Data Analytics) in the experimental analysis. To represent the compute API-based category, we chose the state-of-the-art single GPU-based hardware-conscious join [28], which we refer to as **HCJoin**.

For the GroupBy baseline, we use the most recent GroupBy technique, proposed in [23], which we refer to as **TGB** (Tunable GroupBy). They implement the single attribute GroupBy approach described in[14] using OpenCL, and provide knobs to tune the parameter settings of the code.

For both baselines, we used the original source codes, which are publicly available. For completeness, we also include parallel CPU-based hash join and GroupBy operators [21], which we refer to as **CPU**, as part of the evaluation.

6.2 Graphics vs. Compute

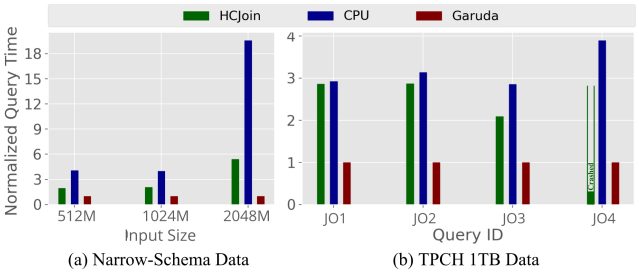
As mentioned in Section 1, the graphics drivers are designed to improve pre-fetching for specific GPU memory types such as vertex buffers. To quantify this improvement, we design the following experiment whose main task focuses on reading data from and writing data to GPU memory. Specifically, we implement, using both CUDA and Vulkan, code that takes as input an array of integers (size = 32M), and computes a histogram where the bins are identified using the LSB-24 bits (i.e., there are 2^{24} = 16M bins). In the Vulkan implementation, the histogram is stored as an image where the MSB-12 and LSB-12 bits of this 24-bit number is used to index a $4K \times 4K$ image. While Vulkan does not require any parameters to be set, the CUDA implementation was evaluated for a large suite of parameter settings. Specifically, our combinations covered: 1) number of thread groups; 2) number of threads per group; and 3) data access pattern (grid stride vs block stride). We then use the L2 cache hitrate metric to compare the pre-fetching/caching performance between the two implementations. In particular, we found that the hitrate achieved by the Vulkan implementation was at least **2X** the maximum hitrate over all the parameters for the CUDA implementation, thus demonstrating the effective pre-fetching strategy of the Vulkan drivers.

Furthermore, a recent work [17] evaluated implementations of standard GPGPU algorithms using Vulkan, and showed that it is on average **1.5X** faster than the CUDA-based implementations. To achieve this, they leverage the improved synchronization support offered by the Vulkan drivers, which transitively points to better thread scheduling, and thus better GPU utilization.

6.3 Join Performance

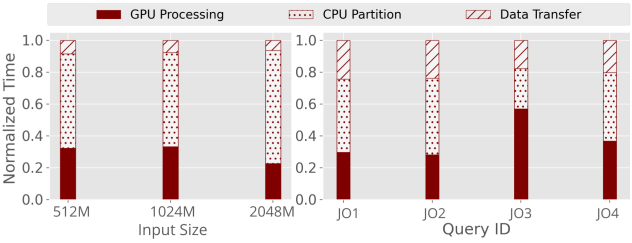
6.3.1 Baseline Comparison. The relative performance of GARUDA, HCJoin, and CPU for the aforementioned narrow-schema data set are as shown in Fig. 8(a), with GARUDA being normalized to 1. GARUDA is at least 2X faster when compared to HCJoin, and 3X when compared to CPU on these relations. The results from the join experiment on the more diverse TPCH tables are as shown in Fig. 8(b). Here as well, GARUDA attains a consistent speedup of at least 2X over HCJoin and around 3X over CPU.

Fig. 9 breaks down the running time of our join approach into three components: CPU partitioning time, GPU processing time, and data transfer time (between the CPU and GPU). Since our implementation pipelines data transfer and GPU processing, the data transfer time does not include the I/O that happens in parallel during GPU compute – here the GPU compute time is the time taken



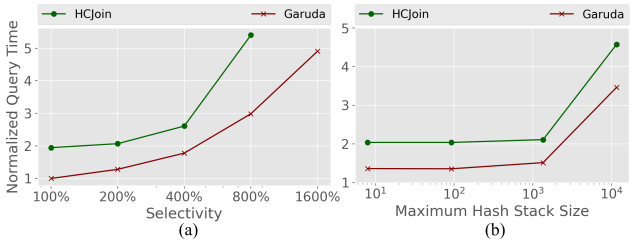
(a) Narrow-Schema Data (b) TPCH 1TB Data

Figure 8: End-to-end Join performance



(b) Narrow-Schema Data (a) TPCH 1TB Data

Figure 9: Join time breakdown of GARUDA



(a) (b)

Figure 10: Impact of Input Data parameters

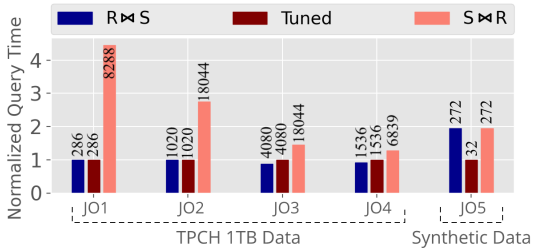


Figure 11: Impact of Snake Join. (Numbers above bars denote hash stack sizes of the join.)

to execute the different rendering pipelines. Due to our pipelined approach, the impact of CPU-GPU transfer overhead is significantly reduced – we obtain a consistent speedup of 2-2.5X using the pipelined approach when compared to a blocking approach.

6.3.2 Impact of Input Data parameters.

Join Selectivity: Fig. 10(a) shows the performance of GARUDA and HCJoin when the build and probe table sizes are fixed to 512M, but the join selectivity (the number of tuples selected relative to the input table size) varies from 100% (output size = 512M) to 1600% (output size = 8192M) on a log-scale. The running times are normalized by setting the GARUDA Join time with 100% selectivity to 1. We see here that GARUDA scales linearly (with a smaller slope) with increased selectivity when compared to HCJoin. Moreover, HCJoin could not even compute the join at 1600%.

Data Skew: Fig. 10(b) plots the normalized running time (normalized w.r.t. Garuda join on uniform data) with varying skew (represented as the maximum required hash stack size, see Section 3.4.4). Specifically, we increase the skew (by increasing the zipf factor, with the maximum zipf factor being 0.5) of the synthetic narrow schema data ($|R| = |S| = 512$ M), and perform a self-join. Note that if only one of the input tables is skewed, then it does not affect the performance due to our snake join approach. Thus, the self-join provides a worst case scenario for our approach. We see here that GARUDA consistently maintains performance improvement over HCJoin even in the skewed regimes. The jump in query times when the hash stack size is 10^4 corresponds to a sharp increase in join output size – here, output size increases by a factor of 4 from around 780M tuples (with stack size 10^3) to 3.16B tuples.

6.3.3 Impact of Snake Join. Fig. 11 analyzes the impact of the partition-based SnakeJoin ordering (Section 3.4.3) when compared to a fixed table-based ordering for JO1–JO4. We observe that an inappropriate join order can significantly impact the running times, becoming up to 4 times slower. On the other hand, our SnakeJoin approach incurs only a marginal overhead when compared to using the ideal join order. Since the initial partitioning time is the same across all algorithms for a given pair of join input relations, we ignore this component in the evaluation.

At first glance, it may appear that the correct choice of join order would have already been made by the query optimizer, and therefore the SnakeJoin is redundant. To demonstrate that there are databases where it can be beneficial despite the optimizer's presence, we also created a synthetic database with tables R and S such that a subset of R has several duplicates in S and is independent of a subset in S which also has several duplicates in R. The join performance on this database is shown in Fig. 11 as JO5. Due to the symmetric nature of the data, the performance of both the *global* join orders is the same – in contrast, SnakeJoin is significantly faster in completing the join processing thanks to its dynamic switches of the build partition across R and S.

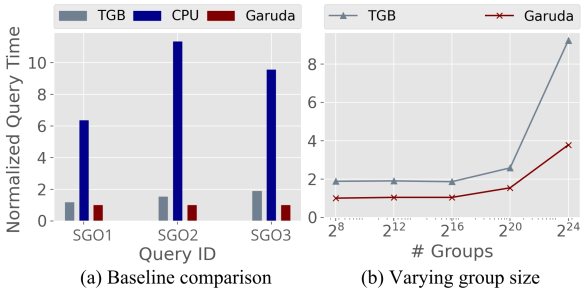
6.4 GroupBy Performance

Since existing GPU-based GroupBy implementations cater only to the base case of a single grouping attribute, we split the evaluation into two parts—single attribute and multi-attribute, respectively.

6.4.1 Single Attribute GroupBy. For this experiment, we exhaustively evaluated the combinatorial number of alternative parameter settings for TGB, as described in [23], and finally chose the settings that resulted in the best query run time. This parameter search was done for all three queries used in our evaluation.

The performance of GARUDA, TGB and CPU are shown in Fig. 12(a) for SGO1 through SGO3. We observe that GARUDA is perceptibly better than TGB, despite the expensive tuning provided to the latter. This highlights how leveraging the graphics pipeline automatically translates to hardware-conscious performance. Both of these GPU-based approaches are significantly better than CPU.

Drilling down, SGO1 results in around 2000 groups covering the input relation of 6 billion tuples. This data characteristic is



(a) Baseline comparison (b) Varying group size

Figure 12: GroupBy performance

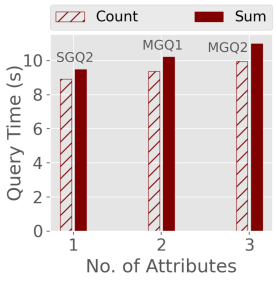


Figure 13: Multi-Attribute GroupBy

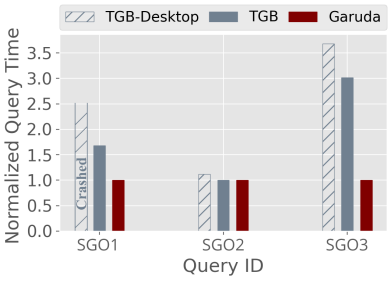


Figure 14: GroupBy on Intel GPU

tailor-made for TGB since it maintains the entire hash table in GPU memory. On the other hand, there are as many as 1.5 billion and 200 million groups in SGO2 and SGO3, respectively. Due to these large group cardinalities, TGB did not have sufficient GPU memory to execute the query to completion. As a workaround for this limitation, we first split the data into memory resident partitions, and computed the GroupBy for each of the partitions. It is for this modified version of TGB that the performance is shown for SGO2 and SGO3 in Fig. 12(a).

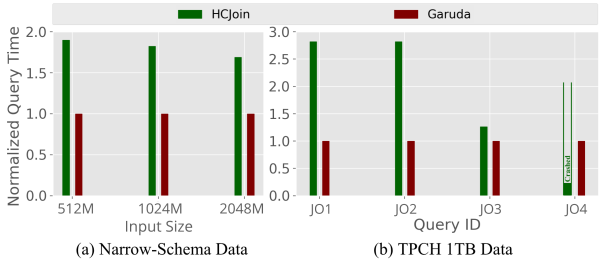
Fig. 12(b) compares GARUDA and TGB with varying group cardinalities. For this experiment, the input relation size was set to 1024M, and the number of groups was varied from 2^8 to 2^{24}.

6.4.2 Multiple Attribute GroupBy. As mentioned previously, only GARUDA is capable of handling multiple grouping attributes. Its performance on the MGO1 and MGO2 is shown in Fig. 13 for both the COUNT and SUM aggregations. For comparative purposes, the performance on the single-attribute SGO2 query is also shown in this graph. We observe here that the queries are all completed in around 10 seconds despite the huge size of the LineOrder relation, testifying to the effective utilization of the hardware resources.

The marginal performance difference between COUNT and SUM is due to the following: For the former, only the grouping attributes have to be transferred to the GPU, whereas for the latter, the value attribute also has to be shipped over. Essentially, the performance difference is dictated by the memory transfer overheads incurred by the grouping and aggregation components of the query.

6.5 Hardware Portability

The goal of this experiment is to show how GARUDA achieves portability essentially "for free" unlike the prior approaches which required expensive customization for achieving their best efficiency. For this purpose, we evaluated the performance on a new hardware platform comprising a laptop with an Intel integrated HD620 GPU. Since the laptop has only 16 GB RAM, the experiment was run on a reduced 100 GB version of the TPCH database. On this new platform, we compared a freshly and fully tuned version of TGB with an unchanged version of GARUDA. The relative running times for the above environment are shown in Fig. 14, which demonstrates that GARUDA cheaply obtains close to tuned performance across hardware architectures. The figure also shows the performance with the parameter settings from the earlier desktop experiments – here we see that for SGO2 and SGO3, there is only a marginal



(a) Narrow-Schema Data (b) TPCH 1TB Data

Figure 15: Join performance on an A100 GPU

10%-20% degradation, but for SGO1, the settings are sufficiently off to cause a crash due to running out of memory.

Since HCJoin uses CUDA, we chose an Nvidia A100 GPU as the alternative hardware to test Join performance. Fig. 15 shows that GARUDA still achieves 2-3X speedup over HCJoin on joins over both the narrow-schema data sets as well as the 1TB TPCH tables.

7 EXTENSIONS AND NEXT STEPS

In this section, we first describe various extensions of the environments modeled in the earlier sections, followed by discussing some of our future work topics.

64-bit join attributes. Equi-joins using 64-bit integer attributes is accomplished by using FBOs with 4-byte color channels instead of the 2-byte channels described in Section 5, as shown in Fig. 16. Specifically, the 64-bit join attribute is split into two 32-bit channels

Figure 16: Pixel layout for 64-bit integer joins.

(g and b). Since, we know that the number of rows processed in a batch is less than 2^{31}, the r channel is sufficient to store the implicit RID, and the a channel is used for the flag. The rest of the implementation remains unchanged.

Non-integer Join Attributes. Thus far, we have only considered integer attributes. However, equi-joins of other data types, such as strings, are also quite common in industrial workloads. We cannot directly represent strings, which can be of significant and variable length, in the graphics data structures (which are of fixed sizes). However, we could leverage the initial partitioning step at the CPU to concurrently also create hashed fixed-length versions of the strings. For instance, using the popular FNV [4] non-cryptographic hash function which is fast, has a low collision rate, and supports

a variety of output lengths, including 32 and 64-bit versions. Of course, the join output may now contain a few false positives due to undetected collisions, but these could be eliminated through a final round of CPU verification of the obtained results.

Graphics pipeline for other operators. While we have shown how the EquiJoin and GroupBy operators can be adapted to leverage the graphics pipeline, extending this approach to other operators may require fresh constructions. Furthermore, it would be interesting to see if a common set of "atomic" pipelines can be designed that can then be composed together to implement such operators.

Integration with query engine. We advocate a CPU-GPU hybrid approach for query processing, since it might not be efficient to move all operators to the GPU. This is especially true when working with data that does not fit in CPU memory, let alone GPU memory. In such instances, it is only natural to take advantage of the already efficient CPU-based operators such as index scans.

However, there exist several challenges for graphics-based GPU operators to be part of a full-fledged query engine, including, when to use GPU operators; how to efficiently schedule and handle movement between the CPU and GPU when these operations are part of a more complex plan (e.g., [5, 6]); how to accurately model the cost of these operators, especially given that hardware configurations can vary greatly across environments; and how to incorporate the operators within the query engine.

8 RELATED WORK

GPUs in the context of database systems have long been explored in the research community. We refer the reader to the recent book [20] and survey [22] for a comprehensive history of the use of GPUs for query processing. In this section, we restrict our attention to GPU-based techniques proposed for the Join and GroupBy operator.

Joins. He et al. [12] was one of the earliest works to propose what has now become the modus operandi for hash joins – i.e. using CUDA. This work also tried using the graphics pipeline for GPU-based joins. Specifically, they implemented the same CUDA-based algorithm using DirectX and showed that the CUDA approach performed better than the graphics pipeline alternative. This lack of performance can be attributed to fitting the graphics pipeline to a GPGPU algorithm, rather than adapting the algorithm to the graphics pipeline. Kaldewey et al. [13] ported the then state-of-the-art CPU hash join algorithms to GPUs to understand the impact of unified virtual addressing (UVA) on the memory transfer bottleneck. More recently, Guo and Chen [11] proposed purely GPU-based joins where they first partition the data in the GPU, and then compute the joins of these partitions in parallel. All of these approaches work only on data that completely fits within GPU memory.

Rui and Tu [25] used new generation GPU features such as atomics, dynamic parallelism, and pipelining data transfer in their hash join approach. Their approach only requires the build table to fit in GPU memory. Sioulas et al. [28] proposed a hardware-conscious hybrid partitioned hash join approach which first partitions the input in CPU, and then executes the join between the partitions on the GPU, thus being able to handle data that does not fit in GPU. Additionally, it also takes advantage of new atomic features offered by the GPU for non-blocking operations.

Rui et al. [24] and Lutz et al. [16] extend the partitioned hash join approach to perform the initial partitioning directly on the GPU. Specifically, they redesign the partitioning algorithm to take advantage of the fast NVlink interconnect between the CPU and GPU. Note that these methods are complementary to our approach: similar to their approaches, the NVlink-based partitioning can also replace our CPU-based partitioning, thereby further improving the efficiency of our approach.

In summary, all of the techniques that use modern hardware are designed using CUDA and follow the traditional GPU compute paradigm. On the other hand, we follow a radically different approach that aims to extract the best performance from the GPU through the use of the already optimized graphics pipeline. Further, the high-level abstractions of our algorithms are amenable to the utilization of macro hardware advances.

GroupBy. Karnagel et al. [14] proposed a CUDA-based GroupBy approach that leverages the modern GPU features such as pipelining and atomic operations, but assumed that the the hash table fits in GPU memory. They also discussed the limitations of hardware portability and choice of parameters. Tomé et al. [30] followed a similar approach, but used perfect hash functions. They also took advantage of the low level GPU architecture by carefully planning local memory placement and use of the GPU registers. However, note that such an approach is not easily portable. Rosenfeld et al. [23] implemented the aggregation approach proposed in [14] using OpenCL, thus being able to evaluate this approach across different GPU hardware. They demonstrated the importance of carefully choosing the hardware parameters, and proposed a systematic technique to choose good parameters for a given query-hardware combination. However, this identification is a computationally expensive process. As shown in our evaluation, designing algorithms using the graphics pipeline can help overcome such limitations.

9 CONCLUSIONS

Our goal in this paper was to resurrect the G in GPU, that is, to leverage the graphics pipeline for implementing OLAP database operators. We showed that relational data could be represented as a stacked sequence of FBO images, and that workhorse operators such as HashJoin and GroupBy could be successfully implemented using core pipeline modules such as Vertex Shaders and Fragment Shaders, in tandem with blend functions and attachments. Our experimental evaluation over industrial-strength warehouse environments on multiple computing platforms showed that GARUDA is capable of providing both improved query performance and highly desirable software engineering features.

Moving forward, we plan to take a two-pronged approach in tackling the open questions highlighted in Section 7: We will explore the design of additional operators (e.g. Projection and Set operators) using the graphics pipeline, and in tandem, work on integrating the already designed operators into the Microsoft SQL Server engine to validate GPU applicability for commercial environments.

ACKNOWLEDGMENTS

We thank Ashish Panwar for his advice on software tools for monitoring GPU performance, as well as Krithika Subramanian and Srinivas Karthik for their valuable feedback.

REFERENCES

[1] Periklis Chrysogelos. 2022. Efficient Analytical Query Processing on CPU-GPU Hardware Platforms. (2022), 132. https://doi.org/10.5075/epfl-thesis-8068

[2] collision 2022. Probability of Collision in Hash Function. https://iq.opengenus.org/probability-of-collision-in-hash.

[3] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 1–8. https://doi.org/10.1145/602259.602261

[4] fnv 2022. Fowler–Noll–Vo hash function. https://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function.

[5] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[6] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *Proc. VLDB Endow.* 13, 6 (mar 2020), 884–897. https://doi.org/10.14778/3380750.3380758

[7] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 325–336. https://doi.org/10.1145/1142473.1142511

[8] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast Computation of Database Operations Using Graphics Processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/1007568.1007594

[9] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. 2005. Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 611–622. https://doi.org/10.1145/1066157.1066227

[10] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* 1, 1 (jan 1997), 29–53. https://doi.org/10.1023/A:1009726021843

[11] Chengxin Guo and Hong Chen. 2019. In-Memory Join Algorithms on GPUs for Large-Data. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 1060–1067. https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00151

[12] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[13] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware* (Scottsdale, Arizona) *(DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[14] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS@VLDB*.

[15] KHRONOS Group 2020. Vulkan API. https://www.khronos.org/vulkan/.

[16] 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*. ACM, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3514221.3517911

[17] Nadjib Mammeri and Ben Juurlink. 2018. VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 25–35. https://doi.org/10.1109/IISWC.2018.8573477

[18] metal 2020. Metal Framework. https://developer.apple.com/documentation/metal.

[19] Microsoft 2017. DirectX programming. https://docs.microsoft.com/en-us/windows/uwp/gaming/directx-programming.

[20] Johns Paul, Shengliang Lu, and Bingsheng He. 2021. *Database Systems on GPUs*. Foundations and Trends in Databases, NOW Publishers.

[21] Mark Raasveldt and Hannes Muehleisen. [n.d.]. *DuckDB*. https://github.com/duckdb/duckdb

[22] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (jan 2022), 38 pages. https://doi.org/10.1145/3485126

[23] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2019. Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3329785.3329922

[24] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (dec 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[25] 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) *(SSDBM '17)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3085504.3085521

[26] 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 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[27] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3* (8th ed.). Addison-Wesley Professional.

[28] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[29] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2003. Hardware Acceleration for Spatial Selections and Joins. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/872757.872813

[30] Diego G. Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. 2018. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing. In *ADMS@VLDB*.