Hardware-conscious Hash-Joins on GPUs

Panagiotis Sioulas*, Periklis Chrysogelos*, Manos Karpathiotakis†, Raja Appuswamy‡, Anastasia Ailamaki*§
*EPFL, †Facebook, §RAW Labs, ‡EURECOM
*firstname.lastname@epfl.ch, †manos@fb.com, ‡raja.appuswamy@eurecom.fr

***Abstract*—Traditionally, analytical database engines have used task parallelism provided by modern multisocket multicore CPUs for scaling query execution. Over the past few years, GPUs have started gaining traction as accelerators for processing analytical queries due to their massively data-parallel nature and high memory bandwidth. Recent work on designing join algorithms for CPUs has shown that carefully tuned join implementations that exploit underlying hardware can outperform naive, hardware-oblivious counterparts and provide excellent performance on modern multicore servers. However, there has been no such systematic analysis of hardware-conscious join algorithms for GPUs that systematically explores the dimensions of partitioning (partitioned versus non-partitioned joins), data location (data fitting and not fitting in GPU device memory), and access pattern (skewed versus uniform).**

In this paper, we present the design and implementation of a family of novel, partitioning-based GPU-join algorithms that are tuned to exploit various GPU hardware characteristics for working around the two main limitations of GPUs–limited memory capacity and slow PCIe interface. Using a thorough evaluation, we show that: i) hardware-consciousness plays a key role in GPU joins similar to CPU joins and our join algorithms can process 1 Billion tuples/second even if no data is GPU resident, ii) radix partitioning-based GPU joins that are tuned to exploit GPU hardware can substantially outperform non-partitioned hash joins, iii) hardware-conscious GPU joins can effectively overcome GPU limitations and match, or even outperform, state-of-the-art CPU joins.

***Keywords*-join, GPU, databases, analytics**

I. Introduction

As the ongoing trend of exponential data growth, combined with an increasing demand for real-time analytics, has created tremendous pressure on existing data analytics systems to scale query execution. As a result, the past few years have witnessed a growing adoption of General Purpose Graphics Processing Units (GPU) as accelerators of choice for data-intensive applications. Despite the growing adoption of GPUs in several domains ranging from numerical analysis to machine learning, it is unclear as to whether GPUs are suitable for accelerating analytical SQL database engines due to the fact that supporting general-purpose join operations on GPUs is surprisingly complicated due to two reasons.

First, analytical queries in warehousing applications are complex in nature and consist of large-to-large joins. Join algorithms used by CPU-based, in-memory analytical engines are optimized based on the assumption that all data is memory resident, and such an assumption usually holds as modern servers can be equipped with Terabytes of memory. However, join algorithms tailored for GPUs cannot make such assumptions, as GPUs are limited in the amount of on-board capacity, with even high-end GPUs being limited to 32GB of device memory. Second, given that data cannot be placed entirely in GPU memory and a paging technique has to be applied for maintaining the working set necessary at the moment, either implicitly through Unified Virtual Addressing (UVA) or explicitly by transferring chunks of data over the PCI-e bus when required. Transfers are expensive because their throughput is bound by the PCI-e bandwidth with a theoretical maximum of 15.8 GB/sec.

Despite these challenges, GPUs provide several benefits compared to CPUs. First, GPUs provide massive parallelism using thousands of cores that can together perform computations at a throughput several orders of magnitude higher than CPUs. For instance, each Nvidia Tesla V100 GPU packs 5,120 CUDA cores packed into multiple streaming multiprocessors (SM) and can deliver 14 TeraFlops of single-precision floating point performance. Second, using tightly integrated High-Bandwidth Memory technologies, GPUs provide device memory with bandwidth approaching 1 TB/sec. Third, unlike their predecessors, modern GPUs also offer a variety of features, like programmable on-chip shared memory that functions similar to a CPU cache with a peak bandwidth of several TB/sec, thread and warp synchronization primitives, techniques for overlapping computation with I/O, all of which provide necessary tools for properly optimized software to overcome the aforementioned bottlenecks.

Recent work on designing join algorithms for CPUs has shown that carefully tuned partitioned join implementations that exploit underlying hardware exhibit excellent scalability and performance on modern multicore servers [1]–[5]. However, there has been no such systematic analysis of hardware-conscious join algorithms for GPUs that systematically explores the dimensions of partitioning (partitioned versus non-partitioned joins), data location (data fitting and not fitting in GPU device memory), and access pattern (skewed versus uniform). In this paper, we bridge this gap in knowledge by designing and implementing hardware conscious hash join algorithms for GPUs. In doing so, we make the following three contributions:

- We present the design of a family of partitioning-based GPU join algorithms that are fine tuned to exploit GPU hardware characteristics. In doing so, we decisively show that hardware-consciousness is important for GPU joins similar to their CPU counterparts, and a naive approach of simply switching a CPU-based join to the GPU will leave substantial processing capacity untapped even if all data is GPU resident. Our GPU join algorithms can process

4.5 Billion tuples/second when data is GPU resident on a standard dual socket server equipped with a single GPU.

- We present, to our knowledge, the first partitioning-based GPU join that scales well for large-to-large joins even if no data is GPU resident. Our out-of-GPU algorithm employs a novel coprocessing approach that performs partitioning on the CPU and join execution on the GPUs to achieve synergistic pipelined execution. In doing so, we show that a one-size-fits-all approach is not suitable for GPU joins, as customizing the join algorithm based on data location is necessary to achieve the best performance and utilization. Our out-of-GPU join can fully exploit PCIe bandwidth and GPU processing capacity to achieve a throughput of 1 Billion tuples/second even if no data is GPU resident. A standard hash join, in contrast, fails to even fully utilize the PCIe bandwidth.
- We present a thorough evaluation that compares our GPU join algorithms with state-of-the-art CPU counterparts and a popular commercial GPU database under several workloads. In doing so, we show how fine-tuned GPU joins can not only outperform CPU joins in all scenarios. We also show how our coprocessing join makes it possible to replace a CPU-only configuration that uses dozens of CPUs with just a handful of CPUs and a single GPU, thereby reducing capital expenses.

II. Background

In this section, we provide an overview of GPU hardware and hardware-conscious, state-of-the-art CPU joins to set the stage for this work[1].

A. GPU Hardware

Each GPU consists of thousands of compute cores organized into units referred as Streaming Multiprocessors. Each such processor has compute cores, a register file, shared memory and caches. The GPU groups a set of 32 threads into a warp, which forms the basic unit of scheduling and execution. All threads within a warp execute the same instruction in lock step but on different data items. Conditional instructions or branches that cause threads to diverge are handled by executing sets of threads, corresponding to each possible execution side of the branch, in sequence. The GPU hardware also has the ability to group multiple memory requests from threads into a single request, thereby reducing the number of memory transactions. However, in order for such coalescing to be possible, data access from threads should follow a sequential access pattern, with each thread accessing a subsequent memory location with respect to its predecessor within the warp.

At the programming level, CUDA exposes the massive data parallel nature of GPUs to the application using the notion of a group of threads referred to as a thread block. Each thread block is executed by a multiprocessor. The variables local to each thread are stored in the register file and are private to the thread. However, the GPU hardware contains on-board Shared Memory which is shared across all threads in a block. Shared memory on GPUs is similar to the caches present on CPUs in that they are KB-sized and provide fast access to data with low latency and bandwidth of several TB/second. However, unlike the non-programmable CPU caches, CUDA exposes Shared Memory to applications to allow for fast data sharing among threads in the same block. At the global level, GPU also contains device memory that is several tens of GBs in size. Due to tight integration and the use of high-bandwidth memory technologies, device memory of modern GPUs provides bandwidth approaching 1TB/second. Still, device memory is substantially slower compared to shared memory.

In this paper, we restrict our attention to discrete GPUs that are connected to the system via the PCIe bus. GPU hardware contains multiple DMA copy engines for bidirectional data transfer between the CPU memory and the GPU memory. CUDA provides applications the ability to overlap transfers with computation through the use of asynchronous memory copy operations that can utilize these DMA engines to fully utilize the PCIe bandwidth. Modern GPUs and recent CUDA environments also provide a rich suite of features for data sharing and thread synchronization. In order to enable data to be shared among threads within the same warp, CUDA provides special warp shuffle instructions. Later in this paper, we show how we use these instructions to accelerate probing in join operations. GPUs have also supported atomic operations for thread synchronization for a long time. In old GPUs based on the Fermi architecture, these atomic operations were supported using a locking mechanism, and thus incurred a heavy performance penalty. However, modern GPUs based on the Maxwell, Pascal, or Tesla architecture implement efficient atomic operations using shared memory. We use these operations for synchronization during the build phase of joins for shared hashtable creation.

B. Hardware-conscious join algorithms

The canonical hash join algorithm, which forms the foundation of modern join algorithms, operates in two phases. In the first phase, referred to as the build phase, the smaller of the two input relations, R, is scanned and a hash function is used to populate a hash table with tuples. The second phase, referred to as the probe phase, then scans the second input relation, S, and probes the hash table for each tuple in S to find matching R tuples. Shatdal et.al [2] examined the cache performance of canonical hash join and observed that when the hashtable is larger than the cache size, almost every hashtable lookup results in a cache miss. As cache misses result in CPU stalls, they lead to underutilization of processing capacity.

In order to solve this problem, they redesigned the hash join algorithm to achieve cache consciousness and proposed a partitioned hash join variant. The partitioned hash join performs a partitioning step first so that the hash table can be partitioned into cache-size chunks that can be stored in cache throughout the processing of each partition. However, a high partitioning fanout may result in TLB misses and incur a

[1]In this paper, we focus on NVIDIA GPUs and use terminology and concepts from the CUDA programming model. However, all concepts and contributions presented in this work apply to AMD GPGPUs as well.

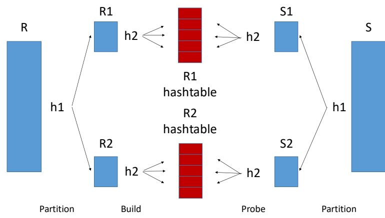

Fig. 1: Anatomy of a partitioned radix join.

performance penalty, because of the high number of different output locations for writing partitions. In order to solve this, Boncz et al propose that multiple partitioning passes of smaller fanouts should be used in order to avoid this performance pitfall [1]. This led to the radix-partitioned hash join algorithm. Figure 1 shows the steps involved in a radix-partitioned hash join example. Both inputs are partitioned using a single pass algorithm. Two TLB entries are sufficient for this example. Hash tables are then built over each partition of input table R and probed for join matches using corresponding S partitions.

III. GPU-conscious partitioned joins

It is well known in literature that partitioned radix hash joins are versatile, scalable algorithms for parallelizing joins on multicore CPUs. Thus, in this section, we start by building a GPU-conscious partitioned radix join for the scenario where all data is GPU resident. In doing so, we show that tuning a join algorithm to be GPU-conscious is a non-trivial task that requires exploring a wide design space consisting with several optimization targets. In Section IV, we will extend the design to cover scenarios where data does not fit in GPU memory.

A. Exploiting shared memory

As we described in Section II-A, modern GPGPUs contain on-board device memory that provides much higher bandwidth than CPU memory. Despite the high bandwidth memory of the GPUs, the effective utilization of the shared memory is important for achieving high overall throughput both because of its speed and its efficiency for more complex access patterns. In order to exploit shared memory for a partitioned join, it is essential that at least one of the working sets in a join fits in the shared memory. Then, the data structure of the build phase is created and stored within the shared memory, allowing for an efficient probe phase that relies on a fast sequential scan of the outer relation and shared memory lookups. The partitioning fanout needs to be high enough to guarantee that the resulting partitions fit in the shared memory allocated for the computation. Then, the join algorithm only needs to consider the pairs of working sets corresponding to the same hash value among the two relations. The intuition behind the partitioned design choice tightly corresponds to the cache consciousness argument in favor of the CPU-based radix hash join and constitutes a parallel among the two algorithms.

The fanout of the partitioning algorithm is, however, also restricted by the amount of shared memory available. The metadata for each output partition needs to be stored in shared memory so that it can be accessed and updated without involving the slower device memory. Also, shuffle space is required so that the threads can rearrange the data before storing it to the final destination, reducing the number of memory transactions due to coalescing. Thus, the partitioning algorithm can have a fanout of at most a few thousand partitions. This restriction constitutes another parallel to the CPU-based radix hash join, in which the fanout is restricted by TLB size [1].

In order to fully exploit shared memory, we use a multi-pass partitioning algorithm that is sized to produce batches of tuples that fit in shared memory. Each pass produces a linked list of buckets per partition. To amortize the overhead of pointer chasing and to improve scan coalescing, each bucket is an array of elements with a capacity that is a multiple of the GPU thread block size. Initially, a pool of buckets is allocated and a subset of them is assigned to the partitions, one per partition, and the threads start storing elements to their respective bucket. When a bucket is full, the partition related to it is assigned a new one from the pool which is linked after the previous bucket. The resulting data structure allows accessing the partition's data in a coalesced access, only following linking pointers to transition among buckets.

For the partitioning steps after the first one, we assign data to CUDA blocks in a round-robin fashion. We experimented with two granularities, a single bucket at a time or a partition at a time (full chain of buckets). The latter approach has the advantage that the CUDA block sub-partitioning a chain of buckets is the only producer of the new partitions and can therefore maintain all the data locally, in the fast shared memory. By contrast, the former processes buckets of different partitions at each step and therefore spends more time initializing internal data structures and accessing data in the GPU memory. However, in case the data is skewed, some of the buckets chains are significantly longer, causing load imbalance and deteriorating the performance of the partition-at-a-time assignment as the longest running CUDA block defines the total execution time. As a consequence, we opt for the bucket at a time assignment, despite the fact that it fares worse for uniform distributions. Long bucket chains produced by the final step are decomposed and assigned to differents Streaming Multiprocessors to balance load during the probing phase.

B. Exploiting warp synchronization

After the working sets have been sufficiently reduced in size, the actual join that consists of a build and a probe phase can be computed. In the build phase the smaller working set is stored in the shared memory, potentially as a data structure that enables efficient equality lookups such as a hash-table. In the probe phase a coalesced scan reads the other working set from device memory and compares the join field to the tuples in shared memory. For this paper, we have implemented nested loop join and hash join kernels for probing.

In the nested loop join implementation, the smaller working set is initially copied to shared memory contiguously. Then, the other working set is scanned in order to compare its elements to the ones in shared memory. Conventional implemen-

```
1   shared_mem ← R partition loaded in shared memory
2   s ← thread private element from (the corresponding) S partition
3   for offset = 0 to size(R partition) with step 32
4     r ← shared_memory[offset + thread_id_in_warp]
5     mask ← ∼0
6     for i in {indexes of bits that may differ, based on partitioning}
7       bit ← 1 << i
8       vote ← __ballot(r & bit)
9       mask ← mask & ((s & bit) ? vote : ∼vote)
10    process matches detected by set bits in mask
```

Listing 1: Detecting matches based on ballot

```
1   def insert(HT, entry)
2     slot ← entry.hash() % #slots
3     old ← atomicExchange(&(HT[slot]), entry.offset())
4     entry.next ← old
```

Listing 2: Wait-free insertion to the hash-table

tations, as in the CPU, would perform all pairwise comparisons individually at this point. However, this implementation is optimized by taking warp-level synchronization primitives and the knowledge that a partition's elements have some common bits into account. Threads read an element of the outer working set and then each warp cooperatively scans the contents of shared memory, as shown in Listing 1. We avoid having each thread read all the values from shared memory by leveraging intra-warp communication to scan 32 values at a time and use ballot[2] instructions to discover matches. Each thread of a warp reads only one of the 32 values from shared memory at each step, line 4. Then, threads iterate over the bits of the values they read and broadcast them using ballot to the other threads, line 6–9. Using bitwise operations, the threads check if the broadcasted bits match the corresponding bit in the value they read from the outer table, line 9. This only needs to be done for the bits not used in partitioning. Therefore, through bitmask manipulations, each thread in the warp compares the value in their registers with 32 inner relation values and retrieves a match bitmask, after only a few ballot operations, reducing the number of memory reads.

C. Exploiting GPU atomics

In the hash join implementation, the smaller working set is stored in shared memory. We store it using a hash-table that uses linked lists for each hash-table slot and use offsets to represents the links between list nodes. The limited size of shared memory allows us to trim the offsets to 16 bits to reduce the memory footprint. The hash table is created in parallel by replacing the reference to the head of each list with a reference to each new element with CUDA's atomic exchange, thus adding the element to the front of the list. After the hash table has been created, the device memory working set is scanned and for each value a lookup is performed using the same hash function. A lookup follows the pointers of chains in order to find the elements with a join field equality. In next sections, this implementation is used by default, because it is more efficient than nested loops as we will see in Section V.

Generating the join output is the final main component of the join algorithm. When coalesced, Device memory accesses are most efficient. Having a different result buffer for each thread is not efficient as it can leave memory unused. Therefore, we buffer the results generated by a warp in shared memory and flush them to the GPU memory sequentially when the buffer is full. At each step of probing their respective chains, the threads of a warp, executed in lockstep, use their synchronization primitives to compute write offsets within their allocated buffer until it is full. Then, the threads flush the buffer's contents to memory, computing the global offset with an atomic increment operation and store any outstanding output that did not fit on the buffer. With this approach, random accesses are avoided. The result of a join operation can be significant in size and materializing it in device memory introduces an overhead, even with coalesced writes.

[2]CUDA's ballot instruction [6] reads a bit from each thread of the warp and aggregates them into a bitmask that is broadcasted to the warp's threads.

IV. Out-of-GPU Execution Strategies

The partitioned join algorithm we described targets GPU-resident datasets. Nevertheless, the amount of data is much larger than the limited device memory of GPUs in practice, even for medium-sized databases. To make things worse, the access patterns of data during the partitioning phase and the scans of the bucket chains are not fit for CUDA's Unified Memory, a mechanism that allows moving pages of memory between the CPU and the GPU on demand, owing to the poor locality of the memory accesses and the fact that only a small portion of a page is needed during an access. Under other circumstances, the Unified Memory could manage the transfers effortlessly achieving throughput levels near the PCI-e bandwidth, which is the upper bound for this case. For the task at hand, a more specialized approach need to be devised, based on asynchronous memory transfers and streams, so as to saturate the PCI-e bandwidth and mask the computation overhead almost completely.

A. Exploiting asynchronous transfers for pipelining

First, we will consider the case that R, the smaller of the two relations, can be stored in the GPU memory. We suppose that the relation has been transferred to the GPU and has undergone the partitioning phase described in Section 3. Then, S, the other relation, can be split to a sequence of chunks that are sufficiently small. Each of those chunks can be transferred to the GPU memory and the join of the chunk with R can be computed using the GPU-resident algorithm. The union of the joins of all the individual chunks with R is equal to the join of S and R, a property that is significant because it provides a way to compute the total result in-GPU despite the fact that the two relations cannot be fully collocated there at any moment.

The use of CUDA streams allows PCIe transfers and GPU execution to occur at the same time. We use one stream for transfers and another for the GPU execution itself, synchronizing tasks on the same chunk with events. We also reserve buffers for two chunks within the GPU, one being transferred and another being processed at each moment. The role of the buffers swaps at each step. Figure 2 shows the pipeline steps as we swap the buffers and the operations that we perform on them. As long as data is transferred over PCIe at a rate

significantly lower than the throughput of the GPU join, there exists an opportunity for performing the full join at transfer speed. In that case, the GPU computational units will be idle until a chunk is available whereas the transfer unit will always be busy because by the time a buffer is picked for use, the previous chunk will have been processed. The execution on the GPU overlaps completely with transfers and is effectively hidden with the exception of the processing for the last chunk. Thus, the total execution time is the transfer time for the data plus the GPU execution time for the last chunk. In our evaluation, we demonstrate that our in-GPU implementation can support such an execution strategy, resulting in near-PCIe bandwidth join throughput when one relation is in CPU.

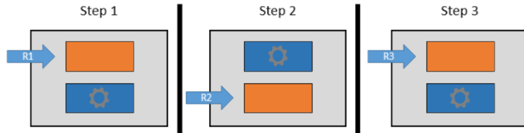

Fig. 2: Pipelining of asynchronous transfers and executions. One chunk is joined while another is being transferred

B. *Exploiting CPU–GPU co-processing*

The other case for the join's input relations is that neither of them can fit in the GPU global memory. In this scenario, the working set for the join algorithm as described cannot be completely GPU resident at any given time even if we stream the larger relation through the PCIe bus. The smaller relation can only be partially stored and as a consequence some of the matching elements will not be in global memory. In addition, relying on techniques such as the UVA is not practical because the irregular access patterns of the join will cause parts of the relation to be transferred over multiple times. This would increase the traffic over the PCIe bus, which has already been a bottleneck for joins with one GPU resident relation.

The problem can be circumvented with another level of partitioning on the host memory. We have already illustrated that a partitioned join can reduce the working set of the join so that the hash table fits in the GPU's shared memory. The same principle applies for the global memory as well. Supposing that the two relations are co-partitioned on the join attribute, all the possible matches of the elements in a partition of a relation are contained in the corresponding partition of the other relation. As long as, the smallest partition fits in the GPU global memory completely, an execution strategy that streams the data through the PCIe can be used to join the individual co-partitions. The overall join results are composed by the results of the individual joins of the co-partitions.

In the general case, the join attribute is not known in advance and the partitions need to be computed at the time of the join. We use a radix partitioning algorithm for multi-core CPUs to partition the relations into co-partitions that fit in GPU global memory. Each of the two inputs is split into chunks and each chunk is assigned to a local-to-data thread which partitions it and produces a list of buckets per partition. After an input relation is consumed, the lists from different threads corresponding to the same partition are concatenated. Then, the co-partitions are transferred over the PCIe bus to the GPU global memory. Once there, the join is computed with the partitioned hash join algorithm proposed in the previous paragraphs. If the aggregate size of two co-partitions is larger than the GPU memory, they are further partitioned.

As data always have to be transfered at least once though the PCIe bus, regardless of the throughput of the partitioning algorithm and the GPU join, the overall throughput is upper bounded by the transfer rate. We match this rate by hiding the computational cost of the CPU and the GPU processing through pipelining. The processing on the CPU side, the memory transfer and the GPU join can be executed asynchronously. We handle the dependencies between memory transfer and the corresponding GPU operations through CUDA's event synchronization, while host-side processing dependencies with the transfers are resolved implicitly by the order of their execution. Dependencies between transfers and GPU operations as well as among GPU operations demand that an operation is done with a buffer before the next one can access it. As long as processing tasks are such that their execution time is lower than that of the corresponding transfer time, the total execution time will be slightly larger than that of the transfer time.

At first, both relations are in host memory in their original order. We partition the smallest relation and store the partitions in pinned memory to allow for faster asynchronous transfers and we initiate the transfer of a working set of partitions to the GPU. Then, we begin the processing pipeline. We subdivide the larger table in chunks that can be streamed through the remaining GPU memory. Then, for each chunk we follow a pipeline of partitioning, transferring the partitions corresponding to the working set and joining the co-partitions within the GPU. These phases overlap between different chunks. A simplified version of the concept is illustrated in Figure 3. In practice, however, we do not have strict pipeline slots. We initiate each operation of the sequence as soon as the previous step is completed. The combined results of the joins between chunks and the working set comprise the overall results of the join for the partitions that the working set consists of. As we show later in Section V, partitioned hash join saturate PCIe bandwidth and does not stall the transfers due to the dependencies. The cost of partitioning is also hidden if the algorithm used achieves a high enough output throughput. Ideally, the fraction of the CPU partitioning output that corresponds to the working set should be produced at a rate higher than that of the PCIe bandwidth so that the transfers do not face starvation and are always ongoing.

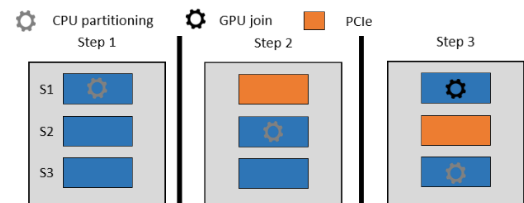

Fig. 3: Pipelining of co-processing steps (partition,transfer,join) in the second out-of-GPU strategy.

After the first working set has been processed, another working set is transferred to the GPU device memory. At this point, all the chunks have already been partitioned and are in pinned memory. As a result, there is no partition phase anymore and the pipeline consists of transfers and joins.

In order to achieve the throughput required for sustaining the transfers to the GPU in a multi-socket system, the partitioning algorithm needs to be NUMA-aware. However, this requires setting the affinity of the data and the results so that the threads perform local accesses during this step. As a result, part of the partitioned data will be located in the far socket with respect to the GPU and have to be transferred through the QPI. In this case, the throughput is usually lower compared to that of transfers from the local socket. Even worse is the fact that when there is congestion in the QPI, even due to the cache coherency protocol, existing traffic interferes with the transfers and their throughput is reduced significantly. To avoid this problem, we transfer the data manually from the far socket to pinned memory in the near socket with CPU threads. We extend the CPU phase of processing in the pipeline with this NUMA-aware copy. Then, this copy constitutes the CPU phase of the pipeline after the first working set.

The overlap of the transfers and CPU processing, and the cache coherency traffic over the QPI, share the resources of the same memory system. In case of intense multi-threading, the memory bandwidth for the near socket may be saturated, causing a collapse in both the transfer throughput and the CPU processing throughput. As a consequence, the overlapping processes need to be configured so that the memory system can support the ongoing operations. Due to the need of maximizing the transfer rate and the fact that the cache coherency protocol cannot be controlled directly, we artificially reduce the memory bandwidth consumed by partitioning. First, we use non temporal hints in order to avoid reading the memory locations used for output. Second, we reduce the number of threads used in the computation to directly alleviate the memory pressure. Based on the expected per-thread memory bandwidth consumption during partitioning , we select the maximum number of threads that allows enough bandwidth for any overlapping data transfers to the GPU to operate at full throughput, as they are on the critical path. This optimization goal is an important factor in configuring the proposed execution strategy. We leave as future work dynamically changing the number of threads during execution.

C. Exploiting multiple DMA engines

GPUs have two distinct DMA engines and support asynchronous transfer both from CPU to GPU and from GPU to CPU at the same time. This technique opens up different possibilities for the management of the pool of data within and outside of the GPU. For the context of this work, we use this functionality to support retrieving the results materialized in GPU memory during the evaluation of the out-of-GPU execution strategies. The common denominator is that another stream is added along with a number of event synchronizations that guarantee both data (i.e. results are available) and resource dependencies (i.e. output buffer is free).

For the first execution strategy, the result materialization mirrors the double buffering on the input side. Two output buffers are required to create a pipeline for output. While, the first buffer is getting filled up by the GPU join kernels, the other one is being transferred to the CPU with an asynchronous transfer. Once the transfer finishes and the partial join is completed, the buffers swap roles. Figure 4 depicts this action relative to the current state of the normal execution pipeline. If the results are at most equal in size with the input, then the cost of transfers to CPU can be hidden through overlapping with the exception of the last transfer. However, if the size of results is larger than the input, then GPU execution stalls due to output buffer dependencies and, as a consequence, input transfers stall due to input buffer dependencies. Then the execution time is determined by the output transfers.

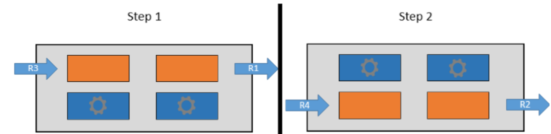

Fig. 4: Pipelining transfers, execution and materialization

For the second execution strategy, the result materialization can be done by adding another phase to the co-processing pipeline for PCIe transfers of the output data. Again we use two alternating output buffers for computing the output and transferring it to the CPU in a pipelined manner. Because of tight memory constraints, it might be the case that the two output buffers are shared with other GPU operations. This introduces extra dependencies for the execution. When transfering the output completes before the buffer is needed again, the pipelined execution hides the output transfers.

D. Handling Skew

The performance of the execution strategy depends on two essential assumptions about the working sets of partitions that are chosen to be processed at each step. First, the size of each working set needs to fit in the GPU memory allocated for the small relation. Second, for the first working set, the transfers to the GPU overlap with the partitioning of the CPU chunks and therefore its size should be large enough to hide the CPU partitioning execution time.

Skew in data results in unevenly sized partitions when an algorithm such as radix partitioning is used. In the case of a skewed smaller relation, a naive choice of working sets can render each of the two assumptions invalid. On the one hand, if too many large partitions are placed in the same working set, the input cannot fit in the buffers. On the other hand, if too many small partitions are placed in the first working set, then the CPU partitioning cannot be hidden completely, causing the transfers to stall until data is available. Therefore, a more elaborate approach to choosing working sets is required.

Our approach of packing partitions into suitable working sets follows two steps. In the first step, a knapsack algorithm

is used to generate the first working set as the set of partitions that maximize the total number of elements, under the constraint that they fit in the allocated GPU memory, padding included. The rest of the partitions are then greedily packed into working sets with the restrictions that each working set fits in the allocated GPU memory and that at most one partition for which space reserved after sub-partitioning exceeds a threshold can be placed in a working set. The idea for the latter restriction is that we quantize the GPU memory in buffers and partitions that exceed the threshold need more space for both partition results and the intermediate results of the first partitioning pass on GPUs.

V. Evaluation

In this section, we describe the experiments for measuring the performance of the methods described above.

A. Experimental Setup

We run our experiments on a Red Hat Enterprise Linux 7.2 machine equipped with two 12-core Intel Xeon E5-2650L v3, 256 GB of main memory and a Nvidia GTX 1080 GPU with 8 GB of device memory. At the time the experiments were run, CUDA 9.0 was installed on the machine.

For the workload, we adopt the one that has been used in several previous studies for evaluating CPU-based joins [3]–[5]. The workload mimics a typical in-memory join processing scenario with two narrow tables, each consisting of a 4-byte key and a 4-byte value stored in a columnar fashion. We use one table as the build and the other as the probe table. As the cardinality of the tables varies per experiment, we describe it inline with each experiment and describe it in Annotations & configurations for Figures 7–13.

We run each experiment multiple times and calculate the execution time as the average execution time of the iterations. We use total throughput, in terms of tuples processed per second by the join algorithm, as our metric of comparison. We compute the total throughput by dividing the combined size of both input relations by query runtime.

B. GPU join processing

Nested loops vs hash join. We first discuss the experiments with both relations stored in GPU memory. In Figure 5, we plot the total throughput of the join as well as the throughput of joining co-partitions for the two join variants described in Section III, a nested loop, implemented using warp shuffles, and a hash join, implemented using GPU atomics. Each CUDA thread block has been configured with shared memory for 2048 elements, 1024 threads and for the hash join with 256 hash table buckets. For this micro-benchmark, each input has two 4-byte columns, a join key and a payload. The keys are unique and uniform. Each thread locally aggregates the output payload columns and at the end attomically updated the global aggregates. We use 2 million tuples per table, vary the number of partitions and plot the results against the partition sizes.

Initially, the nested loop variant has a higher throughput than the hash join one for small partition sizes but the hash join variant outperforms it for larger partition sizes. Until 1024 elements per partition, the throughput of joining co-partitions increases, though for the case of hash join at a higher rate. This is because we utilize the streaming multiprocessor's resources, which have been pre-allocated for 1024 elements, to a greater extent. Then, the throughput of both techniques declines. For hash joins that is because of collisions whereas for nested loops because of its quadratic complexity. The decrease for nested loops is much sharper and is reflected at the total join throughput. Still, the partitioning phase dominates the execution time and the overall difference is small.

Shared vs device memory. In the next experiment, we evaluate the advantages of using shared memory vs device memory for joining the co-partitions. We join two tables with the same characteristics as before, but we vary their size and keep the number of partitions constant. We use two partitioning passes to create 2^{15} partitions and join the co-partitions using a hash-join in either the shared memory or the device memory. We reserve enough shared memory for 4096 elements per partition, 512 threads, 2048 hash table buckets and aggregate the output of the join. In Figure 6 we depict the throughput of joining the co-partitions as well as the total throughput for the two variants. The version that uses shared memory has a higher throughput in probing and consequently the overall join. With shared memory, the throughput of co-partition joining increases as we increase our dataset size because there are more elements per partition and again resource utilization improves. This pattern is traced by the GPU memory versions but the throughput starts declining as soon as chains start to form. The end result is that the shared memory approach is more than 30% faster for the largest relation size. This occurs despite the fact that partitioning accounts for the majority of the execution time.

Annotation & configuration For Figures 7–13, the plots use the annotation that configurations with a concrete line represent an aggregation of the join results, while a mixed pattern of dashes and dots represents result materialization, unless specified otherwise. Similarly, a concrete line means a 1:1 join, a dashed line 1:2 and a dotted line a 1:4 join. In addition, for each CUDA block we allocate enough shared memory for 4096 elements and 2048 hash table buckets. For partitioning kernels we use 1024 threads per CUDA block and 512 threads per block for joining co-partitions. We compare our proposed GPU join strategies, marked *GPU Partitioned* in the plots, with a non-partitioned GPU hash join, *GPU Non-partitioned* and state-of-the art CPU algorithms, the optimized partitioned hash join PRO and non-partitioned hash join NPO presented in [3], [7]. We directly use the source code provided by these studies for the CPU algorithms. As our server is equipped with 24 CPU cores across two sockets, both NPO and PRO fully parallelize execution on all 48 threads.

Impact of materialization. In Figure 7 we evaluate the impact of materializing the join results. We run our proposed algorithm for equally-sized in-GPU relations from 1 million to 128 million tuples and measure the throughput when the results are materialized in GPU memory versus when an

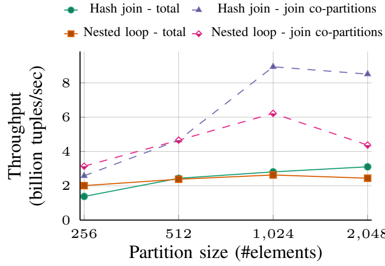

Fig. 5: Comparison of partitioned joins: hash join vs nested loops

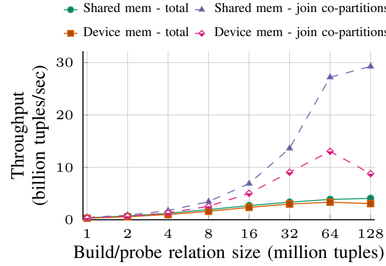

Fig. 6: Comparison of building hash-table in device vs shared memory

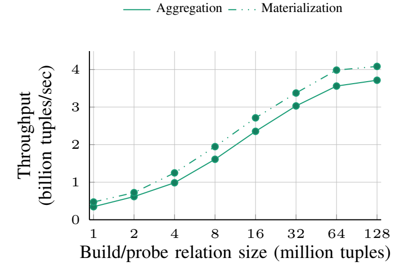

Fig. 7: Partitioned hash join with and without output materialization

aggregate is computed on the payloads. We observe that the version that materializes the output traces the aggregation version for the relation sizes. As both relations have the same distinct values, the selectivity of the join is high and many result tuples are materialized. Still, the overhead of the extra synchronizations for the write buffers and the memory accesses do not degrade performance significantly. Despite the divergence caused by probing the hash-join tables and the occurrence of matches in different cycles for the threads of a warp, our buffering technique manages to combine the result writes and reduce the overhead.

Partitioned vs non-partitioned join. Next, we compare our partitioned hash join implementation with the non-partitioned GPU hash join, in the case that data are already placed in the GPU. For our GPU partitioned hash join, we keep the number of partitions constant at 2^{15} partitions and use the configuration described above. For the non-partitioned GPU join, we use two implementations, a traditional "chaining" implementation that stores the hash table buckets as a chain of elements connected with offset pointers and one with a perfect hash function, as a best-case scenario for the non-partitioning join. For the "chaining" one, when probing the hashtable, three to four random memory accesses are required: one for the hash table itself, one for the key, one for checking that there is no successor in the chain and for the case of a match, an access to the payload. The perfect hashing one is designed to incorporate the knowledge of no-collisions and the contiguous range of unique keys. The payloads of the tuples are stored in a dense array using the keys as offsets. Only one random access is required per probing operation, so the algorithm constitutes a best-case scenario for the non-partitioned hash join on GPUs.

Continuous lines for these three algorithms in Figure 8 depict the throughput achieved by the three algorithms for various dataset sizes that can fit in GPU memory. Initially, the partitioned join performs worse that its non-partitioned counterparts but its throughput benefits from larger dataset sizes as the partitioning overhead get amortized and pays off. As a result, it outperforms the alternatives when the input relations have more than 8 million tuples. By contrast, the throughput of non-partitioned joins starts high but deteriorates rapidly with larger relation sizes.

We repeat the experiment for different build-to-probe table ratios and plot in the same figure the results for a probe relation twice and four times as large as the build relation, with dashed and dotted lines respectively. For each build-side table size, we keep the same set of distinct values in the probe-side, independent of its size, thus the number of matches is increased as we increase the input ratios. The trend is similar to the equally-sized tables, but the improvement of our partitioned method throughput is steeper as, outperforming non-partitioned implementations for even smaller relation sizes.

Payload Sizes Figures 9 & 10 show the impact of the payload size on the performance of partitioned and non-partitioned GPU joins. We use late materialization for retrieving multiple attributes using tuple identifiers as the join payload. We aggregate rather than materialize the payload values because the materialization cost is common. Figure 9 plots the throughput of the two methods for varying probe-side payloads. Non-partitioned joins overtake partitioned ones for larger probe-side payloads due to sequential reads of late materialized attributes, whereas the partitioned join reorders tuples and incurs random accesses. Figure 10 shows the throughput when varying the build-side payload. In this case, both sides reorder the tuples and do random accesses. Thus, the partitioned join maintains its edge. However, as the number of random accesses increases, the difference diminishes.

C. *Out-of-GPU Join*

Streaming probe join. Next, we examine the case that one of the relations cannot fit in GPU memory. We keep the size of the build table fixed at 64 million tuples and vary the size of the probe table keeping its distinct values constant. The probe table is broken down into chunks half the size of the build table. All tables are originally in CPU memory. Figure 11 plots the throughput of the execution strategy for the case that results are materialized on CPU memory as well as when they are aggregated within GPU. The GPU throughput increases with larger probe relations, because the outstanding computations become less significant and comes close to the PCIe bandwidth, the bottleneck for this type of join. Additionally, we notice again that result materialization introduces an overhead but does not cause a significant performance deterioration.

Co-processing join. The final execution strategy for GPU joins in the experiments we performed involves co-processing, which is used when neither of the two relations fits in the GPU memory. The relation sizes vary from 256 million to 1024 million for this experiment. We do not go further than 1024

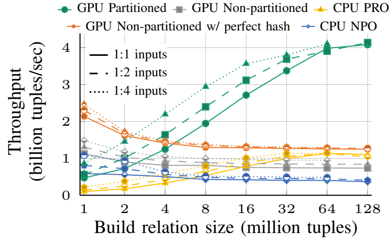

Fig. 8: Comparison of hash join for different build-to-probe ratios

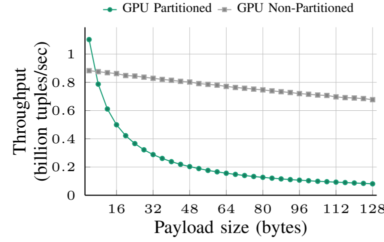

Fig. 9: Effect of varying probe-side payload size

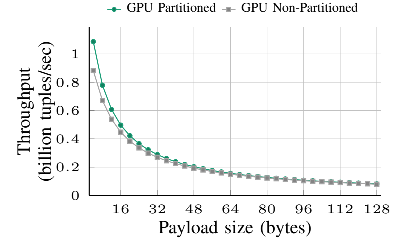

Fig. 10: Effect of varying build-side payload size

million tuples because for the 1:4 relation size ratio, a smaller relation with 2048 million tuples results in a total dataset size of 80GB, leaving insufficient memory space for the CPU-side processing. 16 threads are used for the CPU partitioning phase of the co-processing, as well as software managed buffers with non-temporal memory hints to reduce bandwidth consumption. We perform a 16-way partitioning on the CPU and 5 partitions are used as the working set inside the GPU for the first step. This is because the CPU radix partitioning pass can reach a throughput of approximately 40 GB/s for our configuration and therefore 5 partitions are produced at a rate high enough to saturate the PCIe and still fit in device memory for the largest dataset in our experiments.

The green line in Figure 12 shows the throughput achieved by our co-processing join for different relation sizes and build–probe size ratios. There are two important observations to be made. First, in most cases, the throughput remains insensitive to the relation size. This shows that co-processing is robust for large relations. Second, comparing Figures 7 & 11, we see that our join algorithm provides a throughput of 1.4 Billion tuples/second when only the build table is GPU resident, and 1.2 Billion tuples/second when neither of the tables is GPU resident. Our join algorithms completely saturate the PCIe bandwidth when only the build table sits in GPU memory. As both out-of-GPU cases are bottlenecked on the PCIe, and so we expect that under faster interconnects, like NVLink or PCIe 4.0, our join algorithms would provide higher throughput.

Comparison with state-of-the-art GPU join. In addition we compare our algorithm with two state-of-the-art GPU-enabled analytical engines, DBMS-X, a commercial engine that uses code generation to produce efficient code for GPU-query execution and CoGaDB [8], [9], a research GPU-enabled DBMS system with an operator-at-a-time execution model.

In Figure 14, we show the execution time of our algorithm against the two systems on joining TPC-H [10] tables. We measure the execution time of two joins of the lineitem table, one with the customers table and one with the orders table, and we repeat the experiment for two different scale factors, 10 and 100. For scale factor 10, the first join has a working set of ∼500MB and the second one ∼600MB, excluding any compression that the other systems use. For scale factor we run each query multiple times before running the actual measurements in order to allow the system to load the data into the GPU. Afterwards, all three systems operate on GPU-resident datasets. We observe that our algorithm outperforms both systems. For the scale factor 100, the working sets are ∼5GB and ∼6GB. In joining the customers table, we observe the increase of throughput in our algorithm as well as DBMS-X, which agrees with what in the previous experiments. On the join with the orders table, DMBS-X returns an error, while we revert into the streaming variant of our algorithm which transfers the lineitem table over the PCIe on every query. Unfortunately, CoGaDB was failing to resize an internal data structure while loading scale factor 100.

Figure 15 shows the throughput achieved by our partitioned join and the two systems for equally sized tables. DBMS-X executes the join on GPU-resident data as long as the cardinality falls below 32M tuples. Beyond this, DBMS-X does not load data into GPU memory and simply executes an out-of-GPU join over CPU-memory resident tables. Our join algorithm implementation is able to push this limit to 128M tuples. We suspect this different is due to internal integer size differences in the data type used to represent keys. CoGaDB also manages to reach 128M tuples but it is not designed to operate on joins that do not fit one of the two sides in GPU memory and thus is unable to run the two bigger datasets.

Figure 15 shows that both our algorithms and DBMS-X perform better when data is GPU resident. However, our algorithms outperform DBMS-X in all cases. When data is GPU resident, our algorithms provide a 1.5-2× improvement in throughput over DBMS-X. This difference extends to 10× when data is not GPU resident (right extreme of the graph).

D. Comparison with CPU joins

In this subsection, we compare the GPU join strategies we propose to state-of-the art CPU implementations.

GPU-sized data. First, we examine the case of datasets that fit in GPU memory in Figure 8. For each method, the data is local to the functional unit. We observe that the partitioned and non-partitioned families of joins have the same behavior regardless of hardware. The partitioned joins (PRO and partitioned) improve until they reach a sweet spot while the non-partitioned joins (NPO and non-partitioned) perform well for smaller datasets but not larger ones. Similarly, as the size of the probe table increases, the partitioned join throughput improvement becomes steeper in both cases. It is interesting to note that PRO outperforms the non-partitioning GPU hash join for large enough datasets. This proves inefficiency of the non-partitioned hash join algorithm on GPUs. Second, for all relation sizes, the GPU implementations always outperform

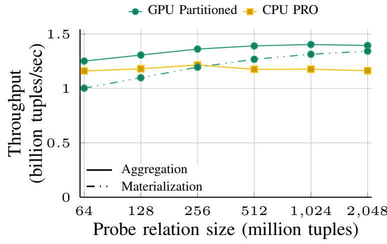

Fig. 11: Streamed probe-side vs CPU

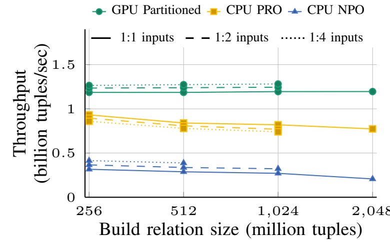

Fig. 12: Co-processing join vs CPU

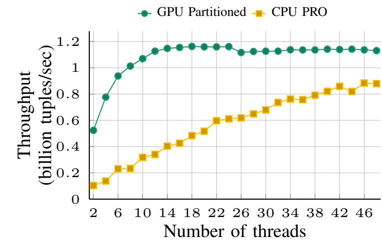

Fig. 13: Scalability with CPU threads

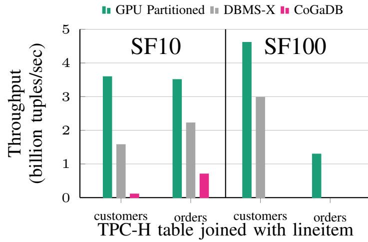

Fig. 14: Joins on TPC-H tables

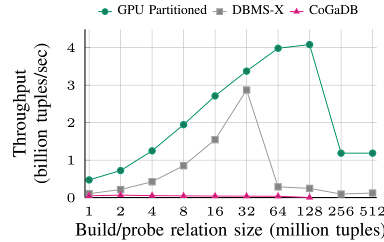

Fig. 15: State-of-the-art GPU systems

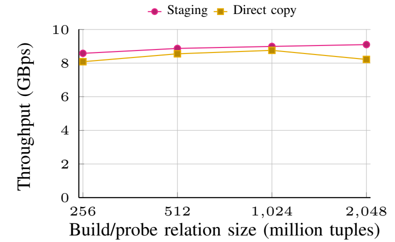

Fig. 16: Staging vs direct copies

their CPU counterparts. For the partitioned joins, the throughput for GPUs reaches as high as 4 billion tuples/sec, a 4x speedup over the CPU version.

Out-of-GPU comparison. Figure 11 compares the probe streaming strategy, when only the build table is in GPU memory, against the CPU partitioned hash join. Figure 12 presents a similar comparison for our co-processing strategy where both tables are out of GPU memory. We see that in all cases the GPU implementation outperforms the CPU join for our experimental setup and in fact the speedup increases with the probe size, approximating the PCIe bandwidth. However, the speedup is still limited compared to the case where all data is GPU resident. Interestingly, we see that the benefit of our co-processing join increases as the dataset size increases because the co-processing throughput remains unchanged while the CPU join throughput follows a downward trend. This is because co-processing is bound by the transfers and the partitioning step, which maintain the same throughput for increasing relation sizes, while for the CPU join the partition sizes increase and the effect of cache optimizations diminish. The discrepancy is even larger when one of the relations is increased in size. This is unlike the probe streaming case which kept the size of the build relation unchanged. This shows that co-processing provides near-PCIe bandwidth performance and scales better than CPU implementations as it maintains a constant throughput.

CPU Utilization. From the above results, it is conceivable that a multi-core CPU with more than 24 cores could match or outperform the GPU execution strategy. Thus, an advantage of using our GPU-conscious join algorithms is that one could achieve the same throughput as CPU-based joins with fewer CPU cores. Figure 13 plots throughput for the CPU partitioned hash join for different numbers of threads against the throughput achieved by co-processing when the same number of threads is used for the partitioning phase. We see that the throughput of the CPU implementation is proportional to the number of threads. On the contrary, the throughput of the co-processing implementation increases rapidly, outperforming the fastest CPU setup with only 6 threads. Co-processing reaches a plateau after 16 threads and faces of small drop after 26 threads. At this point, the memory bandwidth is saturated which decreases the throughput of PCIe transfers that overlap with partitioning. This result shows that using our coprocessing join with a single GPU and 6 cores, we can match the performance of a CPU-based join that uses nearly 10× more CPU cores. This benefit is particular useful for Hybrid Transaction and Analytical Processing Systems (HTAP) like Caldera [11] that use CPUs for transaction processing and GPUs for analytical processing.

NUMA effect In Figure 16, we demonstrate the importance of staging to the near socket before the transfer to the GPU. We plot the throughput for a join of two relations with unique keys that match 1:1 for different dataset sizes and see that performing the intermediate copy improves performance. This occurs because partitioning interferes with transfers from the far socket, deteriorating overall performance.

E. Handling Skew

In Figures 17–20, we examine the performance of our algorithm under skewed data. In Figure 17, we plot the throughput of our algorithms for input tables with 32 million tuples each and different zipf distributions for GPU-resident data. We show the cases with skew only on the probe side, only on the build-side and on both. For the case of both inputs being skewed, we present the worst case of both tables having an identical skew, and the same popular values. For this plot we do not flush the results back to the CPU when they overflow the GPU memory due to data explosion in high skew values, but overwrite them in order to isolate the in-GPU performance. In Figure 18, we produce the same plot for the case of 512 million tuples per input and out-of-GPU data where we use out co-processing algorithm. We include both the case of aggregating and materializing the join results.

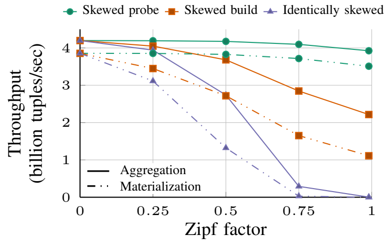

Fig. 17: Skew on GPU-resident data

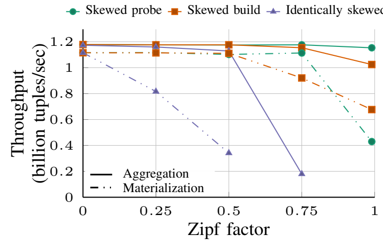

Fig. 18: Skew on CPU-resident data

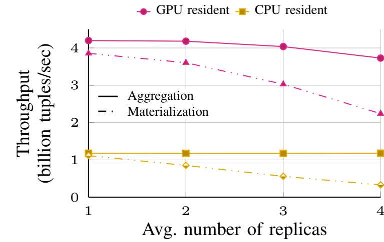

Fig. 19: Uniform number of replicas

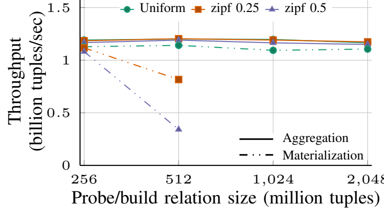

Fig. 20: Input size vs skewed inputs

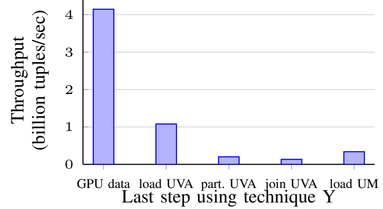

Fig. 21: Effect of UVA and UM

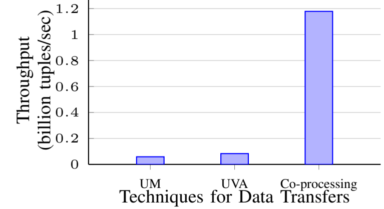

Fig. 22: Throughput with UVA/UM

In Figure 19, we show the impact of uniform distribution with duplicates, for in- and out-of GPU data.

From the first plot we observe that for GPU resident datasets, skew on the probe side has a very low impact on the throughput of the algorithm, if the other side is uniform. Skew on the build-side affects the algorithm more but we still get a throughput higher than the CPU-resident datasets for most of the cases. When the inputs have identical skew, the performance collapses after a factor of 0.75, as the combination of two effects: 1) the build side of the hash-table of each co-partition stops fitting in shared memory and thus we resort to hash-base block nested loops 2) there are too many matches for the very popular values that our algorithm approaches an all-against-all comparison, following long chains for a high percentage of the input. For bigger than GPU memory datasets, we observe from the second plot that our algorithm is much more resilient. As the interconnect is much slower than the in-GPU part of our algorithm, we do not see a performance drop up until a zipf factor close to 1. The GPU-side partitioning and join still perform faster than the PCIe bus. In the case of identical skew, the computation during the probing phase starts to cause overheads that can not be covered by the PCIe bandwidth, after a zipf of 0.75. In these cases, the GPU is affected by the same problem as in the in-GPU case. In the case of result materialization, both in-GPU and out-of-GPU cases observe a small penalty, with the exception of the extreme case of out-of-GPU with identical skew, where the output result explode causing significant volume of data to be written over the PCIe interconnect to the CPU side.

In Figure 20, we show how skew affects the throughput of our co-processing algorithm for different input sizes. We plot the throughput when both inputs have exactly the same distribution and same popular values for zipf factors of 0.25 and 0.5, the worst case, as well as for a uniform distribution. When the results are aggregated, we observe that for zipf factors up to 0.5 there is no performance penalty compared to the uniform case. In addition, as we have already shown in previous experiments, uniform data are also not affected by materialization. Lastly, we observe that for bigger datasets that the output explodes, the performance collapses due to the high amounts of output data.

F. Alternative Data Transfer Mechanisms

Lastly, in Figure 21, we show the throughput for the cases of uniform unique keys when using different data transferring mechanisms, for a working set that fits in GPU memory. In the first figure we show the throughput when the data are GPU resident, as in our in-GPU experiments, the case of using UVA only for loading data, loading and generating the partitions over UVA and executing the whole algorithm over UVA. The last bar shows the throughput using Unified Memory to load the input. In Figure 22, we show the throughput of our algorithm for out-of-GPU data loaded with each of the three methods, UVA, Unified memory and our approach. The first two methods decide on the placement and data transfers, while for the last one we handle them ourselves.

VI. Related Work

GPUs have been extensively used for several decades for accelerating visualization applications from gaming to interactive displays. However, traditional GPUs suffered from several major limitations that made them unsuitable for general-purpose data analytics. First, applications that used GPUs had to manage host (CPU) and device (GPU) memory separately, thus complicating programmability. Second, GPU device memory capacity was too limited to store all data. Therefore, applications had to manually copy data from system to device memory via the slow PCIe bus before executing a computation on the GPU. As a result, despite initial work in late 2000s that showed that GPUs can provide substantial improvement in performance over CPUs [12], [13], they were not widely used in the industry because analytical queries running on GPUs spent most of their time transferring data.

Over the past few years, however, GPUs are evolving from memory-limited accelerators for niche domains to general-

purpose processors with radical improvements along the dimensions of performance due to a continued increase in processing capacity and memory bandwidth, interfacing due to new interfaces like NVLink, and programmability due to the introduction of new functionalities in the CUDA programming model. Thus, there has been renewed interest in designing GPU-based analytical database engines [9], [11], [14]–[17]. However, these systems do not support joins, or use a non-optimized, traditional non-partitioned hash join. In addition, these systems assume that at least one, if not both, relations are GPU resident. In this paper, we present a family of partitioned radix join algorithms that are fine tuned to exploit GPU hardware. In doing so, we present the first study of GPU-based, hardware-conscious join algorithms that can work even if no data is GPU resident.

Kaldeway et al [18] perform a comparison between the GPU implementations of a conventional hash join and a partitioned one and evaluate the effect of UVA. For the traditional GPU approach, in which the input data is copied to GPU memory first, the performance is almost identical. On the contrary, when access is performed over UVA, the conventional approach is superior because the partitioned version requires multiple passes. For the probing phase, the PCI-e bandwidth (6.3 GB/sec at the time) was the bottleneck, whereas for the building phase the computations were the actual bottleneck due to the use of random atomic memory accesses. In this work, we demonstrate that optimized partitioned join algorithms can substantially outperform non-partitioned variants. We also show that our optimized partitioned radix join can saturate the PCI-e bandwidth and offer throughput at least an order of magnitude higher than results reported at [18].

Rui and Tu [19], opt for a two-phase partitioning hash join. They discuss the effect of the features of new GPUs compared to older ones, namely the increased number of cores and registers, the optimized atomic operations, the dynamic parallelism and the overlap of processing and transfers. They also develop a pipelined workflow for processing relations that cannot fit in GPU memory. However, they leave the case of both relations exceeding GPU memory as future work. In this paper, we show how CPU–GPU co-processing allows executing out-of-GPU joins efficiently. In addition, our approach avoids an extra pass on each partitioning step by using GPU atomic operations instead of building histograms.

He et al [20] examine hash joins for coupled CPU-GPU architectures with both units collocated in the same integrated chip, avoiding PCI-e transfers. The authors identify offloading, data dividing and pipelined execution as the main co-processing mechanisms and analyze the hash join as a sequence of fine-grained steps so as to apply the mechanisms. They report significant performance improvements, although the work is specific for coupled architectures. We show that hardware-conscious GPUs joins can provide very high throughput even for discrete GPUs.

VII. Conclusions

In this paper, we implemented a partitioned algorithm for performing joins on GPUs. We demonstrated that for GPU-resident datasets, the algorithm achieves very high performance, with the build and probe step having a throughput close to the CPU's memory bandwidth. Next, we examined the case of out-of-GPU relation joins, more specifically the scenarios in which at least one relation does not fit in GPU, and developed execution strategies. For the former, the larger relation is streamed through GPU, with careful overlaps of memory transfers and kernel execution, whereas for the latter, the host participates in a co-processing scenario, partitioning the relations before streaming them to the GPU. In both cases, the bandwidth of PCI-e has been saturated.

Acknowledgments

This project has received funding from the European Union Seventh Framework Programme, 2013 - ERC-2013-CoG, grant agreement number 617508 (ViDa) and the H2020 - UE Framework Programme for Research&Innovation (2014-2020), 2017 - ERC-2017-PoC, grant agreement number 768910 (ViDaR).

References

[1] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB '99*.

[2] A. Shatdal, C. Kant, and J. F. Naughton, "Cache conscious algorithms for relational query processing," in *VLDB '94*.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proc. VLDB Endow.*, vol. 7, no. 1.

[4] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*

[5] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *SIGMOD '11*.

[6] NVIDIA. Cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[7] S. Schuh, X. Chen, and J. Dittrich, "An experimental comparison of thirteen relational equi-joins in main memory," in *SIGMOD '16*.

[8] S. Breß, H. Funke, and J. Teubner, "Robust Query Processing in Co-Processor-accelerated Databases," in *SIGMOD*, 2016, pp. 1891–1906.

[9] S. Breß, "The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS," *Datenbank-Spektrum*, 2014.

[10] T. P. P. Council, "Tpc-h benchmark specification," *Published at http://www. tcp. org/hspec. html*, vol. 21, pp. 592–603, 2008.

[11] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki, "The case for heterogeneous htap," in *CIDR*, 2017.

[12] G. F. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili, "Relational algorithms for multi-bulk-synchronous processors," in *PPoPP*, 2013.

[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational Query Coprocessing on Graphics Processors," *TODS 2009*, vol. 34, no. 4.

[14] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *PVLDB 2013*.

[15] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "GPU-Accelerated Database Systems: Survey and Open Challenges," *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 2014.

[16] Y. Yuan, R. Lee, and X. Zhang, "The Yin and Yang of Processing Data Warehousing Queries on GPU Devices," *PVLDB 2013*, vol. 6, no. 10.

[17] MapD. https://www.mapd.com/.

[18] T. Kaldewey, G. M. Lohman, R. Müller, and P. B. Volk, "GPU join processing revisited," in *DaMoN*, 2012.

[19] R. Rui and Y. Tu, "Fast Equi-Join Algorithms on GPUs: Design and Implementation," in *SSDBM*, 2017, pp. 17:1–17:12.

[20] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled cpu-gpu architecture," *Proc. VLDB Endow.*, vol. 6, no. 10.