Bandwidth-optimal Relational Joins on FPGAs

Robert Lasch
TU Ilmenau, SAP SE
Walldorf, Germany
robert.lasch@tu-ilmenau.de

Mehdi Moghaddamfar
TU Dresden, SAP SE
Walldorf, Germany
mehdi.moghaddamfar@sap.com

Norman May
SAP SE
Walldorf, Germany
norman.may@sap.com

Suleyman S. Demirsoy
Intel Corporation (UK) Limited
London, United Kingdom
suleyman.demirsoy@intel.com

Christian Färber
Intel Corporation
Walldorf, Germany
christian.faerber@intel.com

Kai-Uwe Sattler
TU Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

The relational equality-join is one of the most performance-critical operators in database management systems. For this reason, there have been many attempts to implement this operator on FPGAs in various sort-merge and hash join variants. However, most achieve suboptimal performance because they ineffectively use the limited memory bandwidths of current FPGA platforms. In this paper, we present an FPGA-based implementation of the partitioned hash join (PHJ), where both PHJ phases are executed on the FPGA. Contrary to prior work, we consider a commonly used PCIe-attached FPGA card with dedicated on-board memory. We discuss how to utilize this on-board memory effectively and propose a solution that uses this memory to store partitioned tuples, minimizing data transfers to system memory and thus optimally using the available bandwidth. In our experimental evaluation we demonstrate up to 2x faster end-to-end join times than state-of-the-art 32-threaded hash join implementations on the CPU.

1 INTRODUCTION

The relational equality-join is among the most performance-critical operators in relational databases and can consume significant memory and CPU resources. Finding efficient CPU-based implementations for the in-memory equality-join has a long history of research [2–4]. The importance of this operator also makes it an attractive target for acceleration using specialized computing devices such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), the latter of which we focus on in this paper.

FPGAs can be integrated into computer systems in different ways, and this significantly impacts how the FPGA can interact with off-chip memory, e.g., *system memory* managed by the CPU. Two common CPU-FPGA architectures are the discrete and the coupled architecture [10]. In a coupled architecture, the FPGA is closely coupled with the CPU, sitting, e.g., in a dedicated mainboard socket [7], and directly interfaces with the CPU to facilitate cache-coherent accesses to system memory. In contrast, a discrete FPGA architecture places the FPGA on a dedicated accelerator card [18], which is attached to the host system through, e.g., a PCIe connection. Accesses to system memory are still possible with this architecture, but need to go through the memory controller. Offsetting this more limited access to system memory, discrete FPGA boards often feature dedicated *on-board memory* which can be directly accessed by the FPGA at the full bandwidth of the memory. Discrete FPGA accelerators cards are more common than coupled architectures and can also easily be retrofitted into existing servers.

For either architecture, the often low bandwidth between FPGA and the host's system memory fundamentally limits how much host-resident data the FPGA can process at any given time [11]. This poses the challenge of using this bandwidth as effectively as possible to maximize the benefit of offloading an operation to the FPGA. Kara et al. [21] and Chen et al. [10] address this challenge on *coupled* FPGA platforms respectively for the partitioning phase and the join phase of the partitioned hash join (PHJ), an implementation variant of the equality-join. In their implementations, partitioned tuples are placed in system memory. This limits the amount of bandwidth available to read input tuples from system memory, or for writing result tuples to system memory, as a significant portion of the bandwidth is already being used to manage the partitioned tuples. Partitioning may also take multiple passes [21], because the size of each partition is not known in advance. This further reduces the effective usage of bandwidth to system memory as tuples then have to be transferred multiple times.

In this paper, we investigate using a *discrete* FPGA platform with dedicated on-board memory for the PHJ. The discrete platform's on-board memory allows us to address the abovementioned challenges by executing both phases of the join on the FPGA and placing partitioned tuples in the on-board memory. This way system memory bandwidth can be used more effectively. However, using the on-board memory also poses new challenges:

(1) It is unclear how to lay out partitions in the memory. Ideally the partitions should be arranged in a way that allows them to grow to arbitrary and different sizes, as this would guarantee that partitioning can be done in a single pass.
(2) The storage of partitioned tuples in on-board memory must satisfy the performance requirements of the partitioner and join components on the FPGA.
(3) On-board memory typically consists of multiple memory *channels*, which each make up a fraction of the available capacity and need to be accessed simultaneously to reach peak bandwidths.
(4) Memory accesses should be mostly sequential to fully exploit the available bandwidth.

Focussing specifically on the PHJ, we show in this paper how a discrete FPGA platform with dedicated memory resources can be effectively utilized for offloading data-intensive DBMS operators. We propose a page management solution (Section 3) that addresses the aforementioned challenges to utilize the on-board memory of a discrete FPGA platform and thus facilitates an FPGA-based PHJ implementation (Section 4) that achieves optimal bandwidth utilization with respect to system memory. We also provide a comprehensive performance model of our design (Section 4.4). Our evaluation (Section 5) shows that the implementation utilizes the available bandwidth to system memory on our test platform optimally and can outperform state-of-the-art multi-threaded in-memory CPU-based join implementations for large input relations. While we demonstrate optimal bandwidth utilization for the specific example of the partitioned hash join (PHJ) in this paper, the techniques presented here may also be more widely applicable to other data-intensive operators, especially ones that also benefit from partitioning and hashing, like aggregation.

2 BACKGROUND

The Partitioned Hash Join. The *partitioned hash join* (PHJ) or GRACE hash join [23] is a hash join that splits the operation into a partitioning and a join phase. It was originally devised to facilitate larger-than memory joins but also has beneficial properties in an in-memory setting [31]. For a hash join in general, tuples from the *build* input relation are first inserted into a hash table based on the join key. The hash table is then probed with tuples from the second input relation (called *probe* relation). Tuples from the probe relation matching the join keys of build tuples in the hash table then produce the join results. For the PHJ, the tuples of both input relations are first partitioned based on the join key. Then, in the join phase, building and probing hash tables is done separately for each pair of partitions, instead of once for the entire input. The PHJ is a good fit for an FPGA-based join implementation, as the partitioning facilitates building hash tables in the fast, but low-capacity on-chip memory of the FPGA.

	Read Volume [B]		Write Volume [B]	
(a)	$r_{partition}$	$= (\lvert R\rvert + \lvert S\rvert) \cdot W$	$w_{partition}$	$= (\lvert R\rvert + \lvert S\rvert) \cdot W$
(b)	r_{join}	$= (\lvert R\rvert + \lvert S\rvert) \cdot W$	w_{join}	$= \lvert R \bowtie S\rvert \cdot W_{result}$
(c)	$r_{partition}$	$= (\lvert R\rvert + \lvert S\rvert) \cdot W$	w_{join}	$= \lvert R \bowtie S\rvert \cdot W_{result}$

Table 1: Read and write volumes between the FPGA and system memory for different join phase placement options.

PHJ Phase Placement. Let us briefly consider the possibilities for placing PHJ phases in a CPU-FPGA setting. Not considering the case where the join is performed fully on the CPU, there are three options which result in different data volumes that *at least* need to be transferred between system memory and the FPGA: **(a)** *partition on the FPGA, join on the CPU* — the approach chosen in [21], **(b)** *partition on the CPU, join on the FPGA* — the approach chosen in [10], **(c)** *partition and join on the FPGA* — which has not been considered in prior work. Table 1 shows the minimal data volumes for the three options, assuming an input tuple width W and a result tuple width W_{result}, and input relations R and S.

To the best of our knowledge, no existing work has attempted to design a bandwidth-optimal (as discussed below) FPGA join solution on a *discrete* FPGA platform. In contrast to coupled platforms, discrete platforms are an interesting target due to their on-board memory, which can be viewed as an additional level in the memory hierarchy and thereby provides opportunities to minimize more expensive accesses to higher levels — i.e., system memory — in the memory hierarchy. The on-board memory gives us the opportunity to choose option **(c)** — executing both PHJ phases on the FPGA. On a *coupled* platform this option would mean that in addition to $r_{partition}$ and w_{join}, $w_{partition}$ and r_{join} would also have to be transferred from and to system memory, which is undesirable due to limited bandwidth to system memory being the bottleneck of current accelerator designs. On the discrete platform however, these data volumes can be handled by the on-board memory by placing the partitions there, making option **(c)** the most sensible choice for an FPGA-accelerated join on a discrete platform.

Bandwidth-Optimality. Any join algorithm that cannot make use of a pre-built index has to read at least $(|R| + |S|) \cdot W$ bytes from memory for its input data and write $|R \bowtie S| \cdot W_{result}$ bytes to memory as results, unless the tuples can be encoded in a more compact way, e.g., using a compression method. Compression has been shown to yield significant performance improvements, e.g., for GPU-based distributed joins, where most time is spent on communication between nodes [13]. This would however add encoding effort on the CPU side, and is thus out of the scope of this paper, as we are looking to construct a purely FPGA-based solution. In the absence of such encoding, an FPGA join system that utilizes the full available memory bandwidth without interruption for the whole duration of the join operation to transfer the abovementioned data volumes from and to system memory cannot be optimized further to improve end-to-end join performance. Better performance is then only possible using a different hardware platform with increased bandwidth. We call an FPGA system that achieves this *bandwidth-optimal*, and demonstrate in Section 5 that our system is bandwidth-optimal under most conditions.

3 FPGA JOIN SYSTEM CONCEPT

In this section we present a bandwidth-optimal FPGA-based partitioned hash join system design for *discrete* FPGA platforms. We first give an overview of the design in Section 3.1. In Section 3.2 we then present the central page management component of the system which is instrumental in being able to utilize the on-board memory effectively and addresses the challenges of using the on-board memory introduced in Section 1.

3.1 System Overview

A high-level overview of our design is shown in Figure 1. The major underlying assumption of the design is that the partitions of the input relations can fit into the on-board memory of the FPGA. This places a hard upper limit on the number of total input tuples, but allows us to use the bandwidth to system memory exclusively for transferring input and result tuples, while the on-board memory is only used for partitioned tuples. For partitioning and joining itself, we adapt existing concepts by Kara et al. [21] and Chen et al. [10], respectively. By coupling these components together with our page management component, we can optimally use the limited bandwidth to system memory.

The arrows in Figure 1 denote the flow of tuples through the system. While red arrows represent the bandwidth limitations of the target platform, blue arrows indicate the processing and transfer rates that the system components need to support to achieve overall bandwidth-optimality. The multiple red arrows between the FPGA and its on-board memory represent the multiple memory channels typically featured on discrete FPGA platforms, which have to be considered to fully exploit the memory. Achieving the target processing rates requires careful dimensioning and design of each of the system components to stay within the resource limitations of the target FPGA. We discuss the target processing rates and bandwidths below, including high-level

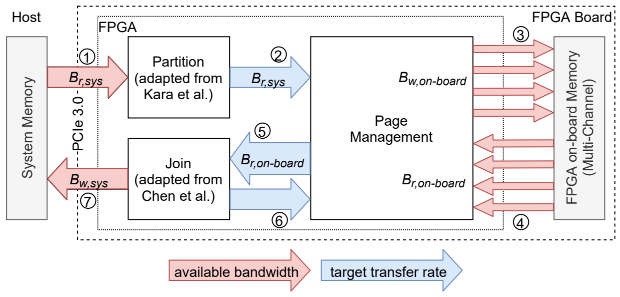

Figure 1: Overview of the proposed FPGA join system.

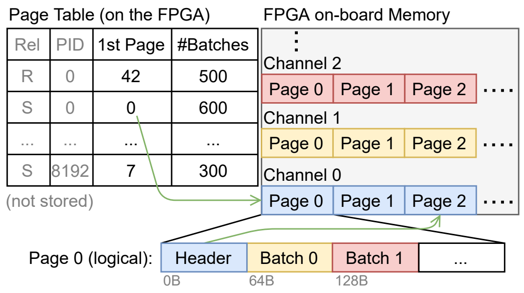

Figure 2: Managing partitions in on-board memory.

overviews of the system components and how they need to be dimensioned to reach the given target rates, which is the key to fully utilizing the system memory bandwidth and thus to a bandwidth-optimal design.

During the *partitioning phase* of the PHJ, input tuples are first read from host memory through ①. The partitioner described by Kara et al. consists of N *write combiners* which group tuples into 64-byte batches that can later be efficiently written to memory. Each write combiner can process one tuple per clock cycle, which results in N tuples per clock cycle total. By adjusting the number of write combiners, the partitioner can be dimensioned to meet throughput requirements. In the original partitioner design, there is also a "Write Back" module that writes the partitioned batches to memory. This is replaced by the page management component in our design, which handles *writing and reading* the tuples to and from on-board memory. In the figure, ② transfers the partitioned batches to the page management component, which writes them to the FPGA's on-board memory through ③. These three interfaces operate at the read bandwidth from system memory $B_{r,sys}$, as this limits how quickly tuples can be transferred from CPU-side buffers and as partitioning does not increase the data volume.

In the *join phase*, partition-by-partition, the page management component sequentially reads build and probe relation partitions from on-board memory (④), and forwards them to the join stage through ⑤. In the join stage, multiple hash tables are built and probed according to the datapath design by Chen et al. [10]. Their design consists of a number of datapaths that each manage a separate hash table. Tuples are assigned to the datapaths according to a datapath partitioning function, which partitions tuples orthogonally to the partitioning done in the partitioning phase. By adjusting the number of datapaths present in the system, higher or lower processing rates can be supported, up to the resource limits of the target FPGA. While probing, the join stage produces join results that are written back to system memory through ⑦. During a build phase, it is also possible that a hash table overflows. In this case, the overflowed build tuples are written back to on-board memory through ⑥ and ③. If an overflow occurs for a partition, the join stage requests the overflowed build tuples and all probe tuples for that partition again from page management after it has finished the first round of probing for that partition. For writing output tuples through ⑦, the system is limited by the available write bandwidth to system memory, $B_{w,sys}$. To saturate this bandwidth, the join stage should ideally accept partitioned input tuples at $B_{r,on-board}$ through ⑤, as this is the maximum read bandwidth of the on-board memory and hence the maximum bandwidth that the page management component can reach reading partitioned tuples from on-board memory through ④.

Note that, depending on the input data to the join operation, either the input side or the output side can become a bottleneck: If few output tuples are produced in relation to the size of the input relations, the input side of the join stage would likely become its bottleneck. If each probe relation tuple matches one or multiple build tuple, the output side of the join stage will become its bottleneck.

3.2 Page management

The page management component needs to address several challenges, which were already introduced in Section 1:

(1) The partitions should be arranged in memory in a way that allows them to grow to arbitrary and different sizes, as this guarantees that partitioning can be done in a single pass.
(2) It needs to accept and supply tuples at rates that allow other system components to reach their target processing rates as discussed in Section 3.1. Concretely, during partitioning, incoming tuple batches with the bandwidth $B_{r,sys}$ need to be written to on-board memory, and while joining, tuples need to be read from on-board memory partition-by-partition as quickly as possible, ideally at $B_{r,on-board}$.
(3) Multiple memory *channels* should be used effectively to be able to read partitioned tuples at the maximum available bandwidth.
(4) Memory accesses should also be mostly sequential to exploit the available bandwidth fully.

To achieve bandwidth-optimality, (1) is most important, as input tuples would need to be read from system memory twice if single-pass partitioning was not possible. We thus employ a *page-based scheme* for managing partitioned tuples in on-board memory, shown in Figure 2: The on-board memory is logically split into equal-sized pages and the tuples of each partition are stored in a singly-linked list of pages. Pages are linked to each other using a page header that stores a pointer to the partition's next page, if one exists.

During partitioning, to write tuple batches to on-board memory, the component keeps track of the current page and a write offset within that page for each partition, and writes the tuples to these offsets as they arrive from the partitioning stage. If, for a partition, the current page is already full or no page has been allocated yet, it is assigned the next free page in memory. This way, arbitrarily many pages (up to the capacity of the on-board memory) can be allocated to each partition, solving challenge (1). While the way partitions are written to memory constitutes a random write pattern, it is not an issue as the write bandwidth is well below the maximum write bandwidth of the on-board memory.

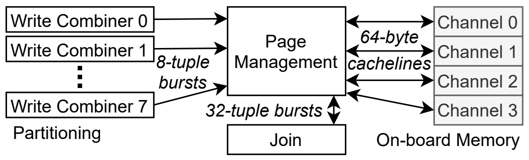

Figure 3: Page management's central role.

To be able to reach the maximum read bandwidth in the join phase, addressing (3), we stripe the logical pages across the physical memory channels. As data is always written to the pages in 64-byte batches, the striping is also done at this granularity. This allows the page management component to read from all memory channels simultaneously to reach the maximum bandwidth when reading a partition. Reading partitions is supported by a page table kept in the FPGA's *on-chip* memory, which stores for each partition the ID of the first page, as well as the total number of tuple batches. As partitions only need to be read sequentially, the singly-linked nature of the page chains is a good fit for this application, naturally addressing challenge (4) , as will be discussed in Section 4.2 in detail.

4 FPGA JOIN SYSTEM IMPLEMENTATION

We now show how to implement the bandwidth-optimal partitioned hash join design presented in Section 3 on the Intel® FPGA Programmable Acceleration Card D5005. Details on this FPGA card can be found in Section 5. For the implementation, we assume a tuple size of 8 bytes, with 4-byte join keys and 4-byte payloads. Result tuples are 12 byte wide, as they consist of the join key and the payloads of both joined tuples. This matches the tuple sizes used in prior work [3, 10, 21]. In the general case of larger tuples, the payload can act as an identifier for a larger tuple kept in system memory (cf. surrogate processing [14]). We discuss implementation details of each of the three system components in the subsections below. As another major contribution, we provide a performance model of the system in Section 4.4 that may be used in a cost-based query optimizer to make offloading decisions. This model can also be used to predict the performance of our system design under different hardware constraints and to guide scaling the design to future hardware with, e.g., higher available bandwidths.

4.1 Partitioning

For the partitioning stage of our system, we port the write combiner design from [21] to OpenCL. In this design, tuples are read from system memory in bursts, and assigned a partition ID using the murmur hash function (see Section 4.3). Each of the read tuples is then forwarded to one of n_{wc} write combiners. The write combiners group partitioned tuples into bursts of eight tuples belonging to the same partition, which are dispatched to the page management component to be written to on-board memory, as shown in Figure 3. On our target platform we can read from system memory at 11.76 GiB/s. To saturate this at a clock frequency of 200 MHz or more, a 64-byte burst needs to be processed in every clock cycle. We therefore use $n_{wc} = 8$ write combiners in our implementation.

4.2 Page Management

The page management component implements the page-based scheme for storing partitions in on-board memory introduced in Section 3.2, and is active during both phases of the PHJ. It is a critical part of our system for achieving bandwidth-optimality. As shown in Figure 3, during partitioning, it accepts bursts of eight tuples from the write combiners of the partitioning stage. One burst is accepted and written to one of the on-board memory channels in every clock cycle. The bursts are accepted in a round-robin fashion. To enable this, per partition, we keep track of the number of bursts already written to memory and the current page used for the partition in local memory, making it possible to quickly determine the destination address for an incoming burst. As shown in Figure 2, a *partition table* stored in local memory keeps track of the number of tuples and the first page ID of each partition.

In the join phase, partitions are requested from page management by the join stage of the system. When a partition is requested, the IDs of the first pages of both the build and probe relation partition are looked up in the partition table. Then, a 64-byte cacheline is requested from each memory channel in each cycle. As our target platform features four on-board memory channels, we read a total of 256 bytes or 32 tuples per cycle, which is 47.68 GiB/s at a clock frequency of 200 MHz. Build relation tuples are sent first to the join stage, followed by probe relation tuples.

Given the linked-list nature of the paging scheme, two factors are important to be able to issue four 64-byte read requests to on-board memory continuously every clock cycle and thus maximize bandwidth utilization: First, we place the page header at the beginning of each page. This way, given a large enough page size, when all cachelines for the current page have been requested, the first cacheline containing the page header has already arrived, making it possible to look up the ID of the next page and to continue issuing requests. In contrast, putting the ID of the next page at the end of each page would cause a gap in the memory read requests every time a page has been fully requested (but not yet fully received) from memory, as the system would first have to wait for the end of the current page to be received from memory to be able to start requesting the next page. Second, the page size must be large enough that the page header arrives from memory before the page's last cachelines are requested. The memory read latency of the on-board memory in our case is in the order of several hundred clock cycles. We hence choose the page size to be such that 1024 cycles pass between requesting the first and last cachelines of a page. Because we read 256 bytes per cycle, this amounts to a **page size of 256 KiB**. Conversely, it is also important to keep the page size as small as possible, to retain flexibility in allocating differently-sized partitions. With the chosen size, the 32 GiB of on-board memory (see Section 5) are split into 131072 possible pages, which is still well enough above the partition count of 8192 to satisfy this constraint.

4.3 Join

The objective of the join stage is to find join results by building hash tables with pre-partitioned build tuples and probing these with the respective probe tuples. Following the design by Chen et al. [10], building and probing is done for multiple tuples in parallel by employing a secondary partitioning of the tuples into *datapaths*. Each datapath builds and probes its own hash table in FPGA BRAMs. We adapt the join stage as follows for our full-PHJ system:

Datapaths. In Chen et al.'s design [10], each datapath can process one tuple every two clock cycles. The authors therefore use 16 datapaths for a processing capability of eight tuples per clock cycle, the number of tuples that can be read from system memory per cycle in their case. We are reading tuples from on-board memory at four times the bandwidth used by Chen et al. Thus the join stage should ideally be able to process 32 tuples per cycle. To this end, we increase the throughput of each datapath to one tuple per clock cycle by applying the forwarding registers technique discussed by Kara et al. [21] for updating hash table fill levels. Ideally, to reach a throughput of 32 tuples per cycle for the whole join stage, we would also double the number of datapaths to 32. However, we were not able to synthesize this number of datapaths: Even though the amount of FPGA resources consumed by the 32 datapaths is well within the bounds of the used FPGA, routing signals and data between central modules and the datapaths becomes a bottleneck and prevents successful synthesis, despite applying further optimizations in the form of sub-distributor and sub-collector modules.

Hash Tables. The hash tables in Chen et al.'s design have a fixed bucket size of four and support no collision handling, which means that if a bucket is full, the respective tuples overflow and need to be handled separately in one or several additional passes. This approach is chosen to minimize the logic resource usage of the hash tables and to support a fixed throughput in each hash table. Due to these benefits, we follow the same approach for the hash tables in our design, but adapt it to guarantee that no expensive overflows can happen for N:1 and near N:1 joins. Such joins are the most common relational joins in practice [4], and we thus specifically optimize for them and also focus our evaluation exclusively on N:1 joins. To optimize for (near) N:1 joins, we size the hash tables such that the hash table buckets, in combination with the datapath partitioning and the partitioning on the input relations, cover the entire 32-bit value space of possible join key values. Concretely, we partition input relations into $2^{13} = 8192$ partitions. This reduces the value space of join keys in *each individual* partition to $2^{32-13} = 2^{19}$ distinct values. When joining, the tuples are again partitioned into n distinct datapaths, which reduces the value space of join keys in each datapath to $2^{19-\log_2 n}$, assuming n is a power of two. We hence size our hash tables to have $2^{19-\log_2 n}$ buckets. For this to work out, all used hash functions have to use different bits of the key value. We shuffle the bits of the key values using the 32-bit murmur hash function [1] and then use a different set of the resulting bits at each of the three steps: The least significant 13 bits of the murmur hash result determine the partition ID for a tuple, the middle $\log_2 n$ bits determine the datapath a tuple is assigned to, and the remaining high bits determine the hash table bucket. This way no overflows can occur for N:1 joins, where the keys in the build relation are unique. Furthermore, as each bucket has four slots, near N:1 joins — where up to 4 tuples with the same key can exist in the build relation — can also be handled without overflows. As all but the $19 - \log_2 n$ highest bits of a tuple key are used to first determine the tuples partition and then its datapath, and hash tables are sized to have $2^{19-\log_2 n}$ buckets, only a single unique tuple key can map to each hash bucket in a given datapath while processing a given partition. Because of this, there is no need to compare tuple keys during probing, as the presence of a tuple in a hash bucket already guarantees that it has the correct key. We thus only need to store *tuple payloads* in the hash tables, which reduces BRAM usage. Note that this optimization may not be possible in general, e.g., if limited resources of the target FPGA prohibit a large hash table size such as the one chosen here. Also note that N:M joins can still be handled by the design by overflowing build tuples and performing one or several additional build-probe passes, but this may carry significant performance costs depending on the tuple distribution. This is an inherent limitation of this hash table design [10]. Our optimzation avoids this limitation for the most common joins, (near) N:1 joins.

Tuple Distribution. Another critical detail of the join stage is the way tuples are distributed to the datapaths. In the original design [10], two distinct mechanisms, *shuffle* and *dispatcher* are used. Shuffle is used to distribute the m ($m = 8$ in [10]) incoming build tuples per clock cycle to the datapaths and to send up to one build tuple per clock cycle to each datapath. For this, each datapath has a single first-in, first-out (FIFO) buffer, which is used to channel the tuples to the datapath and mitigate temporal imbalance in the distribution of tuples among the datapaths. In contrast, the dispatcher mechanism distributes probe tuples to the datapaths using a cross-bar architecture. Without going into further detail, this means that each datapath has m input FIFOs for probe tuples, and thus needs to support probing up to m tuples per clock cycle. One of the implications of this is that the hash tables in the datapaths need to be replicated across several BRAMs to support parallel probing (as a single BRAM only supports one read access per cycle). Additionally, for this mechanism, $m \cdot n$ FIFOs are needed to connect the datapaths to the module that reads partitioned tuples. This is feasible for $m = 8$ and $n = 16$ in [10]. But, in our case m is 32, and n is 16, and we are additionally using larger hash table sizes, making the costs of the dispatcher mechanism design prohibitively high for our implementation. We hence distribute both build and probe tuples using the less expensive shuffle mechanism. The disadvantage of this is that the system becomes sensitive to skew in the probe relation, and we study this effect in our evaluation in Section 5. To reduce high fan-out and fan-in of central modules caused by the many FIFO connections between these modules and the datapaths, we also use a sub-distributor and -collector module between central modules and groups of four datapaths, as also described in [25].

Result Materialization. Finally, efficiently returning result tuples to the CPU also requires special care. This aspect was not discussed in the original join design by Chen et al. [10]. Up to four result tuples can be produced per cycle and per datapath (if a probe tuple matches a completely full hash table bucket). The data volume of the produced results can thereby far exceed the bandwidth available for writing back the results to system memory, and to saturate that bandwidth, result tuples need to be written at a granularity of 64 bytes at a time. In our case, result tuples are 12 byte wide, and we satisfy the granularity requirement by incrementally building bursts of result tuples that can then efficiently be written to system memory. First, we build bursts of eight tuples locally in each datapath. This allows handling the up to four produced results with relatively little logic. For every four datapaths a burst builder module then collects one small burst from one of its datapaths per clock cycle and assembles larger 192 byte wide bursts of 16 tuples. A central module collects the larger bursts and writes one of them to system memory every three clock cycles, saturating the bandwidth if enough results are produced. Each of the aforementioned modules is connected to its predecessor using FIFO buffers, which buffer up to a total of 16 384 results. This allows building a backlog of results in probe phases, when results are produced more quickly than they can be written to memory, and hence facilitates continuously writing results to memory in build phases, when no results are produced.

As each individual burst builder module can collect eight tuples or 96 bytes per cycle (17.88 GiB/s at 200 MHz), the design can also saturate the bandwidth if only one of the modules is active, which could happen when input tuples are heavily skewed and all tuples need to be processed in the same datapath.

4.4 Performance Model

We now develop a performance model for the proposed FPGA join system that can be used to accurately estimate the full end-to-end time required to execute a join operation using the FPGA system. By adjusting its parameters, the model may also be used to predict the performance of the system on other FPGA platforms, e.g., ones with larger host-FPGA bandwidth. The model can further be used to analyze performance bottlenecks and guide the dimensioning of system components (like the number of datapaths) to achieve bandwidth-optimality on potential future platforms.

Parameter	Description	Value/Unit
f_{MAX}	FPGA system clock frequency	209 MHz
L_{FPGA}	FPGA/host communication latency	$\sim$ 1 ms
n_p	Number of partitions	8192
$B_{r,sys}$	System mem. bandwidth (read)	11.76 GiB/s
W	Input tuple width	8 B/tuple
n_{wc}	Number of write combiners	8
P_{wc}	Write combiner processing rate	1 tuple/cycle
c_{flush}	Cycles to flush write combiners	$n_p \cdot n_{wc}$ = 65 536
$B_{w,sys}$	System mem. bandwidth (write)	11.90 GiB/s
W_{result}	Result tuple width	12 B/tuple
$n_{datapaths}$	Number of datapaths	16
$P_{datapath}$	Datapath processing rate	1 tuple/cycle
c_{reset}	Cycles to reset hash tables	1561

Table 2: Summary of parameters used in the model.

Table 2 shows a summary of parameters used in our implementation and for this model. Note that f_{MAX} is the clock frequency of the synthesized FPGA system which will be used in the evaluation. The execution time estimated by the model may for example be used by a cost-based query optimizer to decide for or against offloading a join operation to the FPGA.

Partitioning. In the *partitioning phase*, tuples are processed with the following rate:

$$\begin{aligned} P_{partition,raw} &= \min\left\{ n_{wc} \cdot P_{wc} \cdot f_{MAX}, \frac{B_{r,sys}}{W} \right\} \\ &= \min\{1712\,\text{Mtuples/s}, 1578\,\text{Mtuples/s}\} \qquad (1) \\ &= \frac{B_{r,sys}}{W} = 1578\,\text{Mtuples/s} \end{aligned}$$

As we have dimensioned the system to be able to saturate the available bandwidth $B_{r,sys}$ on our target platform, the second term becomes the limiting factor, and hence the raw throughput of the partitioning stage is 1578 million tuples per second.

However, the total partitioning time is subject to two additional latencies: First, after the partitioning stage has read all input tuples from system memory, it needs to additionally flush the remaining buffered tuples in the write combiners to on-board memory [21]. In our case, this takes up to 65 536 clock cycles, as each of the eight write combiners can have up to 8192 bursts of tuples buffered (one for each partition), and the page management component writes one burst per cycle to on-board memory. This adds a constant latency of $\frac{c_{flush}}{f_{MAX}} = 314\,\mu s$ to the total partitioning time. Second, when invoking the FPGA system from the host code and waiting for it to finish, there is a certain latency associated with the operation, because the OpenCL framework running on the host needs to communicate with the system running on the FPGA through the PCIe bus, which can require multiple PCIe round-trips. In practice, we have observed latencies between 0.8 ms and 1.2 ms, so we model the latency here with the constant $L_{FPGA} = 1$ ms.

In combination, the total partitioning time $T_{partition}(N)$ as a function of the number of tuples N in the input relation is:

$$T_{partition}(N) = \frac{N}{P_{partition,raw}} + \frac{c_{flush}}{f_{MAX}} + L_{FPGA} \qquad (2)$$

Join. For the *join phase*, we focus on N:1 joins and therefore assume that no hash table overflows occur and each tuple needs to be processed only once in the join stage. In contrast to the partitioning phase, where the input and output data volume is equal, for the join phase the number of join results produced in relation to the number of input tuples depends on the join's selectivity. Depending on how many join results are produced, either the tuple processing rate of the join stage's datapaths or the rate at which result tuples are written back to memory become the bottleneck of the join phase. Ideally, input tuples are distributed across all datapaths in the join phase. Thus the number of cycles needed to process n tuples is:

$$c_{p,ideal}(n) = \frac{n}{n_{datapaths} \cdot P_{datapath}} \qquad (3)$$

However, if the keys of the input tuples are skewed, tuples may not be distributed evenly across the datapaths, which can in the worst case result in all tuples being processed by a single datapath. Akin to Amdahl's Law, we model this by assuming that in the case of skew, some fraction α of the tuples can only be processed *sequentially*, using one datapath, while the rest is processed in parallel, using all datapaths. Thus c_p becomes:

$$c_p(n, \alpha) = \frac{\alpha \cdot n}{P_{datapath}} + \frac{(1-\alpha) \cdot n}{n_{datapaths} \cdot P_{datapath}} \qquad (4)$$

How α can be approximated depends on the exact scenario: If it is known that the distribution of tuple keys follows a discrete probability distribution, we have found that α can be approximated well by the percentage of tuples that make up the n_p most frequent values in the relation, which can be obtained using, e.g., the cumulative distribution function in the case of a Zipf distribution. It is conceivable that these most frequent values would dominate the processing time in the case of high skew, as they could in the worst case be distributed across all n_p partitions, thus forming a critical path for the processing. If a histogram of the input relations is available, a scan of the histogram could be done to obtain an approximation of the n_p most frequent values in the input. Lastly, if nothing is known about the input relation, one could obtain a worst-case estimate by setting $\alpha = 1$. Because both input relations can be skewed independently of each other, α exists for both input relations, denoted as α_R or α_S, respectively.

Another significant factor affecting the processing time of input tuples is the time required to reset fill levels of hash tables in the datapaths between each processed partition. Fill levels can be stored using 3 bits each, so we pack 21 of the 32 768 fill levels of each hash table into a 64 bit word. One word can be reset per clock cycle, and hence resetting all fill levels takes

$c_{reset} = \lceil \frac{32768}{21} \rceil = 1561$ clock cycles. This reset is repeated for all $n_p = 8192$ partitions, for a total processing time of:

$$T_{join,in}(|R|, \alpha_R, |S|, \alpha_S) = \frac{c_p(|R|, \alpha_R) + c_p(|S|, \alpha_S) + c_{reset} \cdot n_p}{f_{MAX}} \tag{5}$$

On the output side of the join stage, 12 byte-wide join result tuples are written to system memory. This is limited by the available write bandwidth to system memory, $B_{w,sys} = 11.90$ GiB/s. $T_{join,out}$ is the time required to write all result tuples to system memory and is a function of the number of join results $|R \bowtie S|$:

$$T_{join,out}(|R \bowtie S|) = \frac{|R \bowtie S| \cdot W_{result}}{B_{w,sys}} \tag{6}$$

As either $T_{join,in}$ or $T_{join,out}$ can become the bottleneck of the join phase, the total join phase time with the added latency L_{FPGA} to account for OpenCL overhead becomes:

$$\begin{aligned} T_{join}(|R|, \alpha_R, |S|, \alpha_S, |R \bowtie S|) = \\ \max\left\{T_{join,in}(|R|, |S|), T_{join,out}(|R \bowtie S|)\right\} + L_{FPGA} \end{aligned} \tag{7}$$

End-to-end. Combining the execution time models for the partition and join phases and summarizing common terms yields an *end-to-end model* for the total execution time of a full join operation:

$$\begin{aligned} T_{full}(|R|, \alpha_R, |S|, \alpha_S, |R \bowtie S|) = 3L_{FPGA} + \frac{2c_{flush}}{f_{MAX}} + \frac{W(|R| + |S|)}{B_{r,sys}} \\ + \max\left\{T_{join,in}(|R|, \alpha_R, |S|, \alpha_S), T_{join,out}(|R \bowtie S|)\right\} \end{aligned} \tag{8}$$

Besides the latencies in the first two terms, the third term shows that the system saturates the system memory bandwidth $B_{r,sys}$ during the partitioning phase. If $T_{join,out}$ dominates in the last term, the join phase also saturates the system memory bandwidth.

Let us briefly discuss the expected overheads associated with the integration of the FPGA join with a query engine. As the input to the join is sent and received as a stream of tuples the integration could be implemented similar to an exchange operator known from distributed databases [14]. Any necessary buffering and re-coding could be done in a pipelined fashion with minimal overhead [12].

5 EVALUATION

We now evaluate the bandwidth-optimal FPGA join system presented in Section 4. First, we separately evaluate the throughput of the system's partitioning and join stage, to confirm their bandwidth-optimality. We then compare the system's performance to state-of-the-art CPU-based join algorithms. Although the general case of N:M joins is supported using a basic overflow handling mechanism, we have targeted optimizations towards N:1 joins only and therefore consider those exclusively in this evaluation.

We use a dual-socket system with two Intel® Xeon® Gold 6142 CPUs. These CPUs feature 16 physical cores and 32 threads each, and run at a 2.60 GHz base clock frequency, with a turbo clock frequency of up to 3.70 GHz. The system has 368.5 GiB of system memory and runs CentOS 7.7.1908. Our target FPGA board is the Intel® FPGA Programmable Acceleration Card D5005, which is attached to the system through a PCIe 3.0 16x interface and features 32 GiB of DDR4-2400 on-board memory. The FPGA on this board can access system memory in a shared virtual memory (SVM) model through the PCIe link. In preliminary bandwidth measurements from an OpenCL system running on the FPGA, we have measured peak bandwidths to system memory of 11.76 GiB/s for reading and 11.90 GiB/s for writing, and peak bandwidths to on-board memory of 50.56 GiB/s for reading and 65.35 GiB/s for writing. System and on-board memory can be accessed concurrently at full bandwidth.

BRAM (M20K)	Logic (ALM)	DSP
(66.5%) / 11 721	(66.9 %) / 933 120	(3.8 %) / 1518

Table 3: Resource utilization on the Stratix® 10 SX 2800. DSP blocks are exclusively used for hash calculations.

We compile code running on the CPU using GCC 7.3.1 with optimization level 2 (`-O2`). In our tests, compiling with `-O3` did not yield better, and in some cases even worse results than `-O2`. We restrict CPU join implementations using `numactl` to run on only a single socket to avoid any NUMA effects. The CPU-based join implementations use all available 32 threads of a single socket. To compile the FPGA system, we use version 20.1.0 build 177 of the Intel® FPGA SDK for OpenCL™ with Intel® Quartus® Prime version 18.1.2. We enable the `-hyper-optimized-handshaking` optimization [19] that is available in the OpenCL tool chain for Intel® Stratix® 10 FPGAs and improves the OpenCL system clock frequency (f_{MAX}) at the cost of slightly increased FPGA resource usage. Table 3 shows the FPGA resource utilization of the synthesized system. The OpenCL system runs at an f_{MAX} of 209 MHz.

At the beginning of each experimental run, input relations are stored in contiguous host memory buffers which have either a row- or column-based layout depending on what is supported by the implementation of the respective join algorithm. Our FPGA join system expects a row-based input layout. The 32 GiB of on-board memory capacity limits the combined size of input relations that the accelerator can process, and our experiments are thus limited to this size. In practice, the limitation could be lifted by spilling partition data to host memory when more than 32 GiB are needed for the partitions. We do not implement or evaluate this, as our main focus is to evaluate the case where partitions *do* fit into the on-board memory and the host memory bandwidth can be used exclusively for reading inputs and writing results. Having to read and write partitions in host memory would reduce the performance of the accelerator, as the same limited bandwidth is then used for reading and writing results. Future FPGA platforms may also feature larger on-board memory capacities, and supporting this would be easily possible with the current design.

5.1 Partition and Join Throughput

To confirm that the partitioning and join stage of our join implementation can saturate the available bandwidths from and to system memory, we measure the throughput of both stages in isolation. We report average throughputs of each phase, which we define as $\frac{\text{number of tuples}}{\text{processing time}}$, in Figure 4. The processing time includes the overhead L_{FPGA} of invoking OpenCL kernels on the FPGA from the CPU side. For partitioning (Figure 4a), the number of tuples is the number of tuples in the input relation $|R|$, and input tuples are stored in system memory. For the join phase, we consider throughput both in terms of processed input tuples (Figure 4b) and in terms of produced result tuples (Figure 4c). In the first case the number of processed tuples is $|R| + |S|$, while

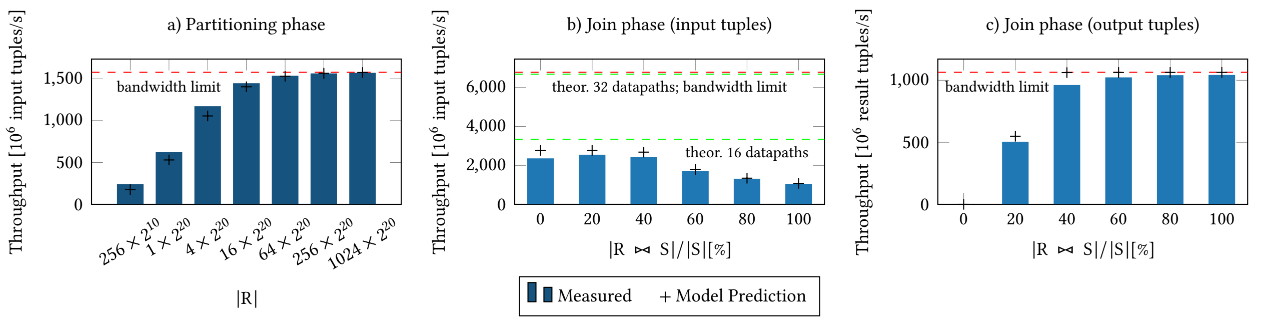

Figure 4: Throughput of (a) the partitioning stage, (b) the input side of the join stage, (c) the output side of the join stage. For (a), we vary the input relation cardinality |R|. For (b) and (c), $|R| = 1 \times 10^7$, $|S| = 1 \times 10^9$ is used, varying the result rate.

in the latter case it is $|R \bowtie S|$. Result tuples are written to system memory. We also report the throughputs predicted by the performance model from Section 4.4.

Partitioning Stage. Figure 4a shows that partitioning throughput grows with the size of the input relation |R|. For small input relation sizes the latencies that we have mentioned in Section 4.4 — flushing the write combiner kernels and overhead for starting OpenCL kernels from the host code — dominate the runtime of the operation. The same latencies become insignificant for large input relation sizes — for sizes larger than 64×2^{20}, the throughput closely approaches the limit given by the available memory bandwidth from system memory of 11.76 GiB/s, which equates to 1578 million tuples per second, shown in the figure as a dashed red line. The predictions of the performance model conform with the measurements relatively well, although the model slightly overestimates the real throughput for relation sizes larger than 4×2^{20}. This is likely due to variance in OpenCL overhead, which we have estimated in the model with a constant 1 ms.

In contrast to Kara et al.'s original design [21], which has a maximum throughput of 514 million tuples per second, the partitioning stage in our system can partition 1578 million tuples per second, as partitioned tuples are not written back to system memory through the same memory interface input tuples are read, but instead are written to on-board memory, utilizing the aggregated bandwidth of our target platform. We have also tested the partitioning stage with constant input relation sizes under varying skew. This does not affect the partitioning throughput, so we omit these results here.

Join Stage. To evaluate join stage throughput, we run the join phase for pre-partitioned input relations that are generated to produce varying numbers of join results. The ratio between input and result tuples is the major factor affecting the throughput of the join stage, as its bottleneck can either be the rate at which input tuples can be processed, or the rate at which output tuples can be written to system memory. As we focus on N:1 joins, where $|R \bowtie S| \leq |S|$, we express the number of results as the ratio $|R \bowtie S|/|S|$, which quantifies the percentage of probe tuples that produce a result tuple. We refer to this as the *result rate*. To produce different result rates, we generate the build relation R to have unique keys from the range $[1, |R|]$ and the probe relation S to have randomly selected keys from a range that is sized such that $|R \bowtie S|/|S|$ becomes the desired value. As the effect of fixed latencies on the observed throughput at small input sizes is already clear from our measurements of the join stage throughput, we chose large input relation sizes here, allowing us to observe the different bottlenecks in the join stage more clearly. Importantly, the time used to calculate the throughput here includes writing all join results to system memory.

Figures 4b and 4c confirm the effectiveness of our system at high result rates of more than 60 %. At these rates, it can be observed in Figure 4c that the system outputs result tuples at a rate of more than one billion tuples per second, saturating the available write bandwidth to system memory of 11.90 GiB/s, represented by the dashed red line. From a result rate of 60 % to 80 % and from 80 % to 100 %, Figure 4b also shows the throughput on the input side of the join stage decreasing accordingly as more results are produced than can be written to system memory. For result rates of 40 % and below, the processing rate of the datapaths becomes the bottleneck.

At the result rates of 0 % and 20 %, the peak processing rate of the system's 16 datapaths becomes the bottleneck of the operation. With 32 datapaths, the throughput on the input side of the join stage could double at such low result rates, as it is not limited by the available bandwidth on the output side when only few or no results are produced. However, 16 datapaths can already nearly saturate the available bandwidth for result rates as low as 40 %, so the system can deliver optimal performance for such join operations.

The maximum theoretical throughput of 16 and 32 datapaths ($n_{datapaths} \cdot P_{datapath} \cdot f_{MAX}$) is shown in Figure 4b by the lower and upper dashed green line, respectively. This shows the significant effect of the latency for resetting the hash table fill levels, which we have discussed in Section 4.4. The latency considerably reduces the peak processing rate of the join stage below what we have assumed when we designed the system, as Figure 4b shows for the result rates of 0 % and 20 %. The attained throughput falls significantly below the theoretical throughput represented by the lower green dashed line at 3424 Mtuples/s. Reducing this latency is an opportunity to improve the end-to-end throughput of the system at low result rates, irrespective of the number of datapaths.

Overall, these results demonstrate the accuracy of the performance model, and show that our system can saturate the available bandwidth from and to system memory in both PHJ phases.

5.2 End-to-End Join Performance

Next, we evaluate the end-to-end join performance of our FPGA system and compare it against three state-of-the-art CPU-based hash join implementations running on 32 threads:

- *The concise array table (CAT) join* by Barber et al. [4]. As no implementation from the original authors is available, we use an implementation by Wolf et al. [33]. For a fair comparison, we change the implementation's tuple size from 16 bytes to

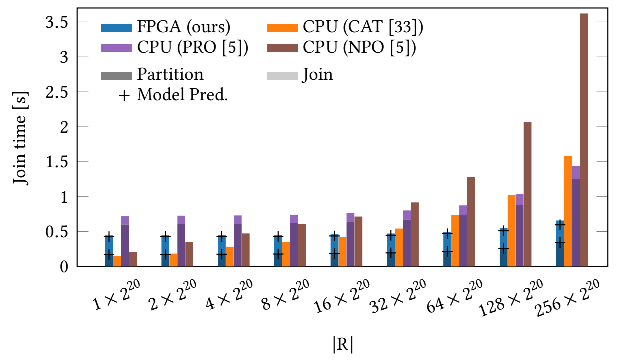

Figure 5: End-to-end join time, varying |R| (|S| = 256×2^{20}).

8 bytes and disable the checksum computation on the result columns.

- *The optimized parallel radix hash join (**PRO**) and the optimized non-partitioned hash join (**NPO**)* by Balkesen et al. [3]. We use the original implementations by the authors [5]. For PRO, 18 radix bits and two-pass partitioning are used.

For the following experiments, input relations are in system memory at the start of the join operation, and time is measured until all join results have been written to system memory. The implementations by Balkesen et al. expect input relations in a row-based input format, and the CAT join implementation expects input relations in a column-based format, so we supply the input relations accordingly. Neither of the CPU join implementations materializes join results in memory, in contrast to our FPGA implementation. Rather, the CPU implementations only compute a total count of the number of join results. We consider this as a reasonable advantage for the CPU, as a join operator executing as part of a query plan could push results directly to subsequent operators (keeping them in the CPU caches) rather than materializing them in memory. This is not possible for a join operator on an FPGA, which necessitates materializing results to memory as an intermediate step.

As we focus on N:1 joins, build relation keys in all following experiments are unordered, dense, and unique, i.e., from the range [1, |R|] — where |R| is the number of tuples in the build relation — and no duplicate keys exist. The payloads for the build and probe relations are generated randomly from the full 32-bit integer range.

Effect of Build Relation Size. We first aim to gain an understanding if and for which problem sizes the FPGA solution can outperform CPU-based approaches. For this, we vary the build relation size |R| at a fixed probe relation size of |S| = 256×2^{20}, with a result rate of 100%. Figure 5 shows the end-to-end join times under these conditions. For the implementations that partition the build and probe relations, the reported end-to-end join times are split into time spent on partitioning and time spent on joining, indicated by darker and lighter colors of the bars, respectively. The predictions of our performance model are also shown for partitioning only and for partitioning and join combined.

A larger |R| naturally causes increased join times for all algorithms, but the performance of the different algorithms scales very differently with increasing |R|. While the FPGA join time at |R| = 1×2^{20} is 2-3 times slower than that of CAT or NPO, the FPGA solution outperforms all CPU algorithms at |R| = 256×2^{20} by approximately a factor of two or more. Amongst the CPU algorithms, the CAT join is the fastest up to |R| = 128×2^{20}, after which PRO performs better than CAT, but CAT also is more sensitive towards increased build relation sizes than PRO. While NPO is nearly on par with CAT for small cardinalities of |R|, its join times increase the most of all algorithms with increasing |R|. This is not surprising, as the performance of non-partitioned joins is known to be more sensitive towards larger build relation sizes because probing larger hash tables quickly becomes more expensive due to cache misses. The results also demonstrate the accuracy of our performance model. Join and partitioning times are accurately predicted, except for the extreme cases where the build relation size exceeds 128×2^{20}, where join time is slightly underestimated. This is because the assumption that there are always enough buffered result tuples to write back during build phases does not hold anymore in these cases, which causes results to be transmitted back to system memory slightly slower than assumed by the model. The $T_{join,out}$ term of the model could be adjusted to account for this, but we consider this unnecessary given the low error caused by this effect.

The FPGA join outperforms all CPU-based joins at build relation sizes of 32×2^{20} tuples and more. It has a significant advantage over CPU-based joins in the join phase of the partitioned hash join, due to the fixed throughput that it achieves for unique build relation key distributions. Looking at the time consumed for the join phase of the FPGA-based solution, one can observe that it is the same for all values of |R|. This is because, in this experiment, the performance of the FPGA solution's join stage is only limited by the system memory write bandwidth, and the total number of join results is the same for all values of |R| in this experiment. The only factor increasing the absolute time used for the FPGA-based join with higher |R| is the partitioning time, which increases because the total number of partitioned tuples increases and with it the time to transfer the tuples from system memory.

It should be noted that the FPGA join system exploits the full available bandwidth to system memory in both phases of the partitioned hash join, except for small inefficiencies with the partitioning at smaller build relation sizes that we have already observed in our analysis of the partitioning stage throughput in Section 5.1. Thereby our FPGA system reaches the optimal performance that is attainable for any FPGA-based join on this particular hardware platform.

We have also conducted experiments with varying probe relation sizes at fixed build relation sizes and found that the relative performance between CPU and FPGA does not change with the probe relation size. This means that the major factor affecting the relative performance of our FPGA- and the CPU-based joins is the size of the build relation |R|, with our FPGA system outperforming CPU-based solutions for |R| ≥ 32×2^{20}.

Effect of Probe-Side Skew. Next, we analyze the effects of a skewed probe relation key distribution. The experiment uses a workload equivalent to **Workload B** used by Chen et al. [10] (|R| = 16×2^{20}, |S| = 256×2^{20}). To evaluate different levels of skew, keys in the probe relation S are generated following a Zipf distribution, varying the Zipf exponent z between 0 and 1.75 in increments of 0.25. It should be noted that $|R \bowtie S| = |S|$ still holds here, as build relation tuple keys follow a unique distribution in [1, |R|] and the skewed probe tuple keys are generated in the same range.

Figure 6 shows the results of the experiment. Without any skew, the performance of the CAT join is roughly on par with our FPGA system. This is identical to the result at |R| = 16×2^{20} in Figure 5. With increasing z, and thereby increasing skew, the

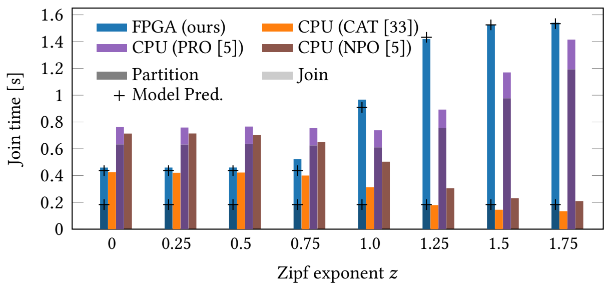

Figure 6: End-to-end join time for Workload B ($|R| = 16 \times 2^{20}$, $|S| = 256 \times 2^{20}$), varying probe-side skew.

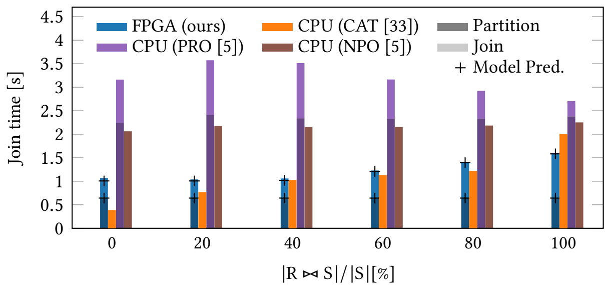

Figure 7: End-to-end join time, varying result cardinality ($|R| = 1 \times 10^7$, $|S| = 1 \times 10^9$).

performance of our system deteriorates, although it remains relatively stable below $z = 1.0$. A similar performance degradation can be observed for the CPU-based PRO, while NPO and CAT exhibit better performance with increasing skew. This is a known effect of the characteristics of the different CPU-based implementations [3]. Thereby CAT and NPO outperform our FPGA system at high skew levels.

As alluded to in Section 4.4, α_S is obtained by evaluating the CDF of the Zipf distribution at n_p. This predicts the real-world behavior under skew well, but further work is necessary to confirm that this works equally well for other distributions or using histograms.

Effect of Join Result Size. To investigate the effect of the number of produced join results on the end-to-end performance, we generate the input relations in the same way as in Section 5.1. Figure 7 reports end-to-end join times including partitioning for these input relations and compares against CPU-based join times. First considering the results of the FPGA system only, the time spent in the partitioning phase does not change at the different result rates. This is expected as the absolute number of tuples that needs to be partitioned does not change with the result rate. In contrast, the join times decrease with decreasing result rates. This is a result of the join stage of the FPGA system being less limited by the memory bandwidth to system memory when emitting result tuples as fewer results are created, as we have already discussed in Section 5.1. Likewise, the performance of the join stage does not improve further when going from a result rate of 20 % to 0 % because the join stage is already at the peak processing rate attainable using 16 datapaths at a result rate of 20 %. The performance of the join stage at low result rates would improve with 32 instead of 16 datapaths. However, this would have relatively little influence on the end-to-end join time, as the end-to-end time is already dominated by the time spent on partitioning at result rates of 20 % and 0 %. As in the previous experiments, the performance model accurately predicts the real-world behavior of the system for these input characteristics.

Comparing the FPGA system to the CPU-based solutions, the FPGA outperforms PRO and NPO in all cases, and the performance of PRO and NPO is also mostly constant across all result rates. In contrast, the join time of the CAT join drops further with each decrease in result rate. At $|R \bowtie S|/|S| = 0\,\%$, the join time drops to 21 % of the time at $|R \bowtie S|/|S| = 100\,\%$. Because of this, CAT outperforms the FPGA solution slightly for result rates below 100 %, and is even two times faster than the FPGA at a result rate of 0 %. This is because the CAT join creates a bitmap that marks all existing keys of build tuples during the build phase. In the probe phase, this bitmap is used to prune tuples early that cannot have a matching build tuple [4], saving memory accesses to the tuple payloads and reducing processing time significantly.

5.3 Discussion

We have shown our FPGA-based system to be competitive with state-of-the-art hash-join implementations on the CPU [3, 4]. Especially at large input sizes, the FPGA outperforms the CPU.

Limitations. In general, the implementation *achieves the goal of fully exploiting the available bandwidth to system memory* on this platform in both phases of the PHJ, with just three exceptions:

Small input sizes: For small input sizes the total join time is dominated by fixed latencies such as resetting hash table fill levels and the latencies involved with managing OpenCL kernels, deteriorating throughput. Our performance model (Section 4.4) can guide the decision for or against offloading in practice.

Skew: Skewed input relations can negatively impact join stage throughput, as a majority of tuples may be routed only through a single datapath at high skew levels, making it a bottleneck. For probe relation tuples, where this is more problematic than for build relation tuples, this deteriorated throughput is due to our decision to remove the dispatcher mechanism from the join stage that was present in the original system designed by Chen et al. [10]. In practice, we find that only very highly skewed key distributions significantly deteriorate the performance of the system.

Too few join results: If the *result rate* is too low, the processing rate of the join stage becomes the bottleneck and thus reduces the rate at which results are written to system memory. This may yet be improved by increasing the internal throughput of the systems' join stage, by e.g., scaling up the system to 32 datapaths. But this would make little difference in practice, as the partitioning time dominates the end-to-end join time when few results are produced anyway. The partitioning time is limited by the available bandwidth, and thus cannot be further improved without additional CPU-side processing, e.g., compressing input data before transferring it.

Outlook. Besides reducing data volumes, future platforms with higher bandwidth that use, e.g., PCIe 4.0 to attach the FPGA, seem most promising to further improve partitioning as well as join performance. We have demonstrated the quality of our performance model in this evaluation, which can also be used to predict performance for higher-bandwidth platforms by changing its parameters. Taking the example of PCIe 4.0, which features double the bandwidth of our current platform, the model would tell us that end-to-end join performance can be doubled by just scaling the number of write combiners in the partitioner from eight to 16, as the 16 datapaths in the join stage would still be able to saturate the bandwidth with result tuples if no skew is present. To support even higher bandwidths, the datapaths would also have to be scaled up, likely also requiring a future FPGA to have more available resources.

6 RELATED WORK

Before discussing related work on join processing using FPGAs, we also take a look at joins using CPUs and GPUs. Finally, we analyze and discuss the hybrid FPGA-CPU join by Chen et al., and compare it against our design.

6.1 Joins on Other Hardware Platforms

CPUs. Efficient implementations of the relational join were extensively discussed in the research literature. We refer to [14, 29] for excellent overviews on this topic. As larger caches and DRAM became available on modern multi-core processors, tuning implementations to these hardware characteristics received special attention [2–4]. We have compared against these implementations as the fastest ones we are aware of. While we focus on join processing on a single node, the efficient execution of joins in a distributed setup is also subject of prior research [6].

GPUs. GPUs also provide an interesting acceleration platform for joins. Fundamental designs of join algorithms on GPUs are discussed by He et al. [17]. Kaldewey et al. [20] utilize NVIDIA's Unified Virtual Addressing (UVA) for a hash join implementation that does not need to fit all data into the scarce GPU device memory. Because UVA extends the memory pool by also using host memory, the bottleneck is ultimately the PCIe bandwidth for host memory access, resulting in an overall throughput of 2-6.1 GB/s. Another partitioned hash join is presented by Sioulas et al. [32]. They among other things evaluate the impact of the input data location for the join. While they achieve throughputs of 4 billion tuples per second for GPU-resident data, for CPU-resident data only 1.2 billion tuples per second are possible due to the PCIe bottleneck. We also consider the case of CPU-resident data and achieve similar throughput, which shows that GPUs suffer from limited interconnect memory bandwidth just like FPGAs. Lutz et al. [24] analyze and evaluate the most recent NVLink interconnect technology for join processing on GPUs. For an optimized no-partitioning hash join the authors achieve a speedup of up to 18× over PCIe 3.0. This also shows that GPU-based join accelerators can benefit greatly from faster interconnects, which is in line with our observations for FPGA-based accelerators. Gao et al. [13] show that a distributed GPU-based hash join can be scaled to 1024 GPUs and achieve an impressive throughput of 1.8 trillion input tuples per second, using lightweight compression to reduce data transfer volumes. Both the use of compression and considering distributed contexts may be promising avenues for future work also for FPGA-based joins.

While GPU-based acceleration approaches achieve outstanding performance, a distinct advantage of FPGAs over GPUs is the ability to create deep on-chip processing pipelines, which helps to avoid off-chip memory data movement. FPGAs allow full flexibility for the implemented circuit, while GPUs have fixed instruction sets and memory hierarchies. This makes FPGAs more power-efficient. As interconnect technology for FPGAs is likely to improve in the future, like it already has for GPUs, it makes sense to at least consider FPGAs as a more power-efficient accelerator alternative to GPUs, even if GPUs currently provide superior performance.

6.2 Joins on FPGAs

Windowed Joins. Several works have explored using FPGAs for time- and energy-efficient stream processing [26] and in this context also for windowed join operations [27, 28]. Here we target full (un-windowed) relational equi-joins.

Relational Joins. A wide range of prior work on relational join processing using FPGAs exists [11, 30]: Halstead et al. [15, 16] present a non-partitioned hash join on the multi-FPGA Convey-MX platform and report throughputs of around 620 million tuples per second using a combined memory bandwidth of 76 GB/s. Casper and Olukotun [8] implement a sort-merge join on another multi-FPGA platform with a combined memory bandwidth of 115.2 GB/s, and reach a throughput of 800 million tuples per second. They however assume that input data is already stored in FPGA on-board memory. Chen and Prasanna [9] further expand the sort-merge approach by partially sorting data on the CPU and performing final sorting and merging on FPGAs. This hybrid approach yields better bandwidth utilization but overall lower throughput as they target a lower-end FPGA platform than previous solutions. Kara et al. [22] perform first experiments with high bandwidth memory (HBM) as FPGA on-board memory. Their hash join implementation can process 80 GB/s if data is already in HBM, but throughput falls to 10 GB/s if data needs to be loaded from host memory first. Peak throughput can also only be reached for small build relations of up to only a few ten thousands of tuples.

A common theme amongst all these works is that join throughput in relation to the available bandwidth is relatively low, i.e., the available memory bandwidth is used ineffectively. This is because data often needs to be transferred from and to memory multiple times to perform the full join operation. We have shown that high join throughput can be achieved even on a relatively low-bandwidth platform (PCIe 3.0) if efficient bandwidth-utilization is considered throughout the design process. For this we utilized existing bandwidth-efficient FPGA-based concepts for the two PHJ phases: Kara et al. [21] implement the partitioning operation on the *coupled* HARP v1 FPGA platform where the FPGA can communicate to the host memory through a cache-coherent QPI interface. Although this design can in theory process 12.78 GB of input data per second, the QPI interface on this particular platform only offers a bandwidth of around 6.5 GB/s bidirectionally, which thus becomes the bottleneck of the design. As partition buffers are allocated in system memory and the FPGA cannot dynamically control their size, their design may also have to fall back to two-pass partitioning if a partition exceeds the pre-allocated size, further reducing throughput. In our design, both problems are solved by using a faster PCIe interface and being able to dynamically allocate partitions in the on-board memory of the discrete FPGA platform. On a similar coupled platform (HARP v2) Chen et al. [10] implement a partitioned hash join where they rely on input that is partitioned on the CPU. We discuss and compare their design to ours below.

6.3 Comparison to Hybrid

The join solution proposed by Chen et al. executes the partitioning phase of the PHJ on the CPU, and uses a novel design on the FPGA for the join phase. Chen et al. do not include partitioning on the FPGA based on a roofline analysis of their target platform, which shows that partitioning on the CPU is more efficient due to limited FPGA memory bandwidth. We adopt their FPGA-based join phase with modifications, but also include partitioning on the FPGA.

On our target platform, partitioning on the FPGA makes much more sense than on the coupled HARP v2 platform, for *two main reasons*: First, partitioning can be executed much more efficiently on the FPGA than on the CPU, as the on-board memory of the FPGA makes it possible to partition in a single pass, reducing both bandwidth and computation requirements. Chen et al.'s decision to place the partition phase on the CPU is largely based on the memory access requirements for the partitioning, as the CPU has higher-bandwidth access to memory. Single-pass partitioning drastically reduces the required memory accesses. Second, a hybrid implementation would perform inferior to the FPGA-only one, because the FPGA part of the system would have to simultaneously read partitioned tuples from host memory and write result tuples back to host memory during the join phase. This is because the FPGA-based partitioner can place partitions directly in the FPGA's on-board memory, while a CPU-based partitioner places partitions in host memory. Therefore the throughput of the join phase would be severely limited in comparison to the FPGA-only solution, as the full PCIe bandwidth can only be used unidirectionally by the FPGA on our target platform. The slower join phase could potentially be offset by better partitioning performance on the CPU, but we find that comparing our FPGA-based partitioning results from Section 5 to the CPU-based partitioner used by Chen et al. [10], shows similar partitioning performance for both solutions.

Besides the major conceptual difference of choosing an FPGA-only approach, we have also discussed implementation changes in comparison to Chen et al.'s solution in Section 4. As an important factor we discuss how to materialize join results back to host memory, which was not included in the original design [10]. When directly comparing the Hybrid results [10, Figure 6b] to ours, which is possible for the result at $|R| = 16 \times 2^{20}$ and $|S| = 256 \times 2^{20}$ in Figure 5 (Workload B in Chen et al.'s evaluation) we observe two things: First, the partitioning time is practically equivalent, second, the join phase runtime is 30 % lower for the hybrid solution. The first reaffirms our decision to include the partitioning into the FPGA system on this target platform, while the latter is a result of the higher available bandwidth on HARP v2, as well as the lack of result materialization to host memory in the hybrid design. In contrast to the hypothetical "FPGA-only" results included in Chen et al.'s evaluation, our solution exhibits much better partitioning performance, as already discussed. Finally, it is important to note that offloading a join operation to our system takes the entire load of executing that operation off of the CPU and frees it up to perform arbitrary other tasks, which is not true for the hybrid solution.

7 CONCLUSION

In this paper we have developed a bandwidth-optimal partitioned hash join (PHJ) implementation on a PCIe-attached FPGA board with dedicated on-board memory. To our knowledge this is the first bandwidth-optimal solution on such an FPGA platform. We optimize usage of the limited bandwidth to system memory by using the FPGA board's on-board memory to store partitions and utilizing a novel paging scheme to enable single-pass partitioning of tuples read from system memory. We have shown in our evaluation that the FPGA system can saturate the available PCIe bandwidth in both PHJ phases, partitioning 1.6 billion 8-byte tuples per second, and processing build and probe tuples at up to 2.8 billion tuples per second in the join phase, writing back up to 1 billion result tuples per second to system memory. Comparing against three state-of-the-art 32-threaded CPU-based join solutions using common workloads has shown that if the build relation cardinality is larger than 32 million tuples, the end-to-end join time of our FPGA system is faster than all CPU-based variants, unless inputs are heavily skewed or the join is particularly selective. We have also presented an accurate performance model of the FPGA system, which considers skew as a factor and can be used to decide for or against offloading a join to the accelerator at query-optimization time. The model parameters can be easily adjusted to make predictions about the performance of the design on other/future FPGA platforms, e.g., ones using PCIe 4.0 for higher host-FPGA bandwidth.

REFERENCES

[1] Austin Appleby. 2016. MurmurHash family of hash functions. https://github.com/aappleby/smhasher. Accessed: 2021-08-27.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and Tamer M Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[4] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, G Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364.

[5] Claude Barthels, Cagri Balkesen, Gustavo Alonso, Jens Teubner, and Tamer M Özsu. 2013. Source code of reference [2]. https://systems.ethz.ch/research/data-processing-on-modern-hardware/projects/parallel-and-distributed-joins.html. Accessed: 2021-08-27.

[6] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528.

[7] Nicholas Carter. 2015. Call for Proposals: Intel-Altera Heterogeneous Architecture Research Platform Program. http://sigarch.hosting.acm.org/2015/01/17/call-for-proposals-intel-altera-heterogeneous-architecture-research-platform-program/. Accessed: 2021-08-27.

[8] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 151–160.

[9] Ren Chen and Viktor K Prasanna. 2016. Accelerating equi-join on a CPU-FPGA heterogeneous platform. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 212–219.

[10] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA Useful for Hash Joins?. In *CIDR*.

[11] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. 2020. In-memory database acceleration on FPGAs: A survey. *The VLDB Journal* 29, 1 (2020), 33–59.

[12] 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*. 1603—-1618.

[13] Hao Gao and Nikolay Sakharnykh. 2021. Scaling joins to a thousand GPUs. In *12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*.

[14] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[15] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins. In *CIDR*.

[16] Robert J Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. 2013. Accelerating join operation for relational databases with FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 17–20.

[17] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2008. Relational joins on graphics processors. In *SIGMOD*. 511–524.

[18] Intel. 2021. Intel® FPGA Programmable Acceleration Card D5005. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-d5005/overview.html. Accessed: 2021-08-27.

[19] Intel. 2021. Intel® FPGA SDK for OpenCL™ Pro Edition: Programming Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf. Accessed: 2021-08-27.

[20] Tim Kaldewey, Guy M. Lohman, Rene Müller, and Peter Benjamin Volk. 2012. GPU join processing revisited. In *DaMoN*. 55–62.

[21] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 433–445.

[22] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High Bandwidth Memory on FPGAs: A Data Analytics Perspective. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[23] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of hash to data base machine and its architecture. *New Generation Computing* 1, 1 (1983), 63–74.

[24] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1633–1649.

[25] Mahmoud Mohsen, Norman May, Christian Färber, and David Broneske. 2020. FPGA-Accelerated compression of integer vectors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–10.

[26] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: a query compiler for FPGAs. *Proceedings of the VLDB Endowment* 2, 1 (2009), 229–240.

[27] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2013. Flexible query processor on FPGAs. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1310–1313.

[28] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2019. Scalable Multiway Stream Joins in Hardware. *IEEE Transactions on Knowledge and Data Engineering* 32, 12 (2019), 2438–2452.

[29] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The Complete Story of Joins (in HyPer). In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, 31–50.

[30] Philippos Papaphilippou and Wayne Luk. 2018. Accelerating database systems using FPGAs: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 125–1255.

[31] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. 1994. *Cache conscious algorithms for relational query processing*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[32] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious Hash-Joins on GPUs. In *ICDE*.

[33] Florian Wolf, Michael Brendle, and Georgios Psaropoulos. 2018. Submission to the SIGMOD Programming Contest 2018: FloMiGe. https://db.in.tum.de/sigmod18contest/downloads/flomige.tar.gz. Accessed: 2021-08-30.