FPGA-based Data Partitioning

Kaan Kara Jana Giceva Gustavo Alonso
Systems Group, Department of Computer Science
ETH Zürich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Implementing parallel operators in multi-core machines often involves a data partitioning step that divides the data into cache-size blocks and arranges them so to allow concurrent threads to process them in parallel. Data partitioning is expensive, in some cases up to 90% of the cost of, e.g., a parallel hash join. In this paper we explore the use of an FPGA to accelerate data partitioning. We do so in the context of new hybrid architectures where the FPGA is located as a co-processor residing on a socket and with coherent access to the same memory as the CPU residing on the other socket. Such an architecture reduces data transfer overheads between the CPU and the FPGA, enabling hybrid operator execution where the partitioning happens on the FPGA and the build and probe phases of a join happen on the CPU. Our experiments demonstrate that FPGA-based partitioning is significantly faster and more robust than CPU-based partitioning. The results open interesting options as FPGAs are gradually integrated tighter with the CPU.

1. INTRODUCTION

Modern in-memory analytical database engines achieve high throughput by carefully tuning their implementation to the underlying hardware [4,5,21]. This often involves a fine-grained partitioning to cluster and split data into smaller cache size blocks to improve locality and facilitate parallel processing. Recent work [31] has confirmed the importance of data partitioning for joins in an analytical query context. Similar results exist for other relational operators [27]. Unfortunately, even though it improves the performance of the subsequent processing stages (e.g., in-cache build and probe for the hash join operator), partitioning comes at a cost of additional passes over the data and is very heavy on random data access. In many cases, it can account for a significant portion of the execution time, nearly 90% [31]. Since it is an important, yet expensive sub-operator, partitioning has been extensively studied to make it faster on modern multi-core processors [3,27,32] and to explore possible hardware acceleration [37,41].

SIGMOD '17, May 14–19, 2017, Chicago, Illinois, USA.
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035946

In this paper we explore the design and implementation of an FPGA-based data partitioning accelerator. We leverage the flexibility of a hybrid architecture, a 2-socket machine combining an FPGA and a CPU, to show that data partitioning can be effectively accelerated by an FPGA-based design when compared to CPU-based implementations [27].

The context for the work is, however, not only hybrid architectures but the increasing heterogeneity of multi-core architectures. In this heterogeneity, FPGAs are playing an increasingly relevant role both as an accelerator per se as well as a platform for testing hardware designs that are eventually embedded in the CPU. One prominent example of this is Microsoft's Catapult which uses a network of FPGA nodes in the datacenter to accelerate, e.g., page rank algorithms [28]. Other such hybrid platforms include IBM's CAPI [34] and Intel's research oriented experimental architecture Xeon+FPGA [24]. The latter one, which we use in this paper, brings an FPGA closer to the CPU enabling true hybrid applications where part of the program executes on the CPU and part of it on the FPGA. Especially because of its hybrid nature, this type of architecture enables the exploration of which CPU intensive parts of an application can be offloaded to an FPGA as it has been done for GPUs [25].

Our contributions are the following:

- The hash function used for partitioning plays a crucial role but robust hash functions are expensive [29]. In our FPGA design we show how to use the most robust hashing available with no performance loss.
- The partitioning operation can be implemented as a fully pipelined hardware circuit, with no internal stalls or locks, capable of accepting an input and producing an output at every clock cycle. To our knowledge this is the first FPGA partitioner to achieve this and improves throughput by 1.7x over the best reported FPGA partitioner [37].
- We compare our FPGA based partitioning with a state-of-the-art CPU implementation in isolation and as part of a partitioned hash join. The experiments show that the FPGA partitioner can achieve the same performance as a state-of-the-art parallel CPU implementation, despite the FPGA having 3x less memory bandwidth and an order of magnitude slower clock frequency. The cost model we developed shows that, in future architectures without such structural barriers, FPGA-based partitioning will be the most efficient way to partition data.

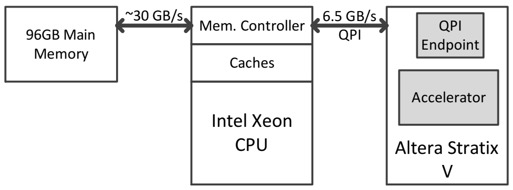


Figure 1: Intel Xeon+FPGA Architecture

2. XEON+FPGA ARCHITECTURE

2.1 General Description

Our target architecture is the *Intel-Altera Heterogeneous Architecture*[1] [24] (Figure 1). It consists of a dual socket with a server grade 10-core CPU (Intel Xeon E5-2680 v2, 2.8 GHz) on one socket and an FPGA (Altera Stratix V 5SGXEA) on the other. The CPU and the FPGA are connected via QPI (QuickPath Interconnect). The FPGA has 64B cache line and L3 cache-coherent access to the 96 GB of main memory located on the CPU socket. On the FPGA, an encrypted QPI end-point module provided by Intel handles the QPI protocol. This end-point also implements a 128 KB two-way associative FPGA-local cache, using the Block-RAM (BRAM) resources. A so-called Accelerator Function Unit (AFU) implemented on the FPGA can access the shared memory pool by issuing read and write requests to the QPI end-point using physical addresses. Our measurements show the QPI bandwidth to be around 6.5 GB/s on this platform for combined read and write channels and with an equal amount of reads and writes. In Figure 2, a comparison of the memory bandwidth available to the CPU and the FPGA is shown, measured for varying sequential read to random write ratios, since this is the memory access characteristic that is relevant for the partitioning operation. We also show the measured bandwidth when both the CPU and the FPGA access the memory at the same time, causing a significant decrease in bandwidth for both of them.

Since the QPI end-point accepts only physical addresses, the address translation from virtual to physical has to take place on the FPGA using a page-table. Intel also provides an extended QPI end-point which handles the address translation but comes with 2 limitations: 1) The maximum amount of memory that is allocatable is 2 GB; 2) The bandwidth provided by the extended end-point is 20% less compared to the standard end-point. Therefore, we choose to use the standard end-point and implement our own fully pipelined virtual memory page-table out of BRAMs. We can adjust the size of the page-table so that the entire main memory could be addressed by the FPGA.

The shared memory operation between the CPU and the FPGA works as follows: At start-up, the software application allocates the necessary amount of memory through the Intel provided API, consisting of 4 MB pages. It then transmits the 32-bit physical addresses of these pages to the FPGA, which uses them to populate its local page-table. During runtime, an accelerator on the FPGA can work on a fixed size virtual address space, where the size is determined by the number of 4 MB pages allocated. Any read or write access to the memory is then translated using the page-table. The translation takes 2 clock cycles, but since it is pipelined, the throughput remains one address per clock cycle. On the CPU side, the application gets 32-bit virtual addresses of the allocated 4 MB pages from the Intel API and keeps them in an array. Accesses to the shared memory are translated by a look-up into this array. The fact that a software application has to perform an additional address translation step is a current drawback of the framework. However, this can very often be circumvented if most of the memory accesses by the application are sequential or if the working set fits into a 4 MB page. We observed in our experiments that if the application is written in a conscious way to bypass the additional translation step, no overhead is visible since the translation happens rarely.

[1]Following the Intel legal guidelines on publishing performance numbers we want to make the reader aware that results in this publication were generated using preproduction hardware and software, and may not reflect the performance of production or future systems.

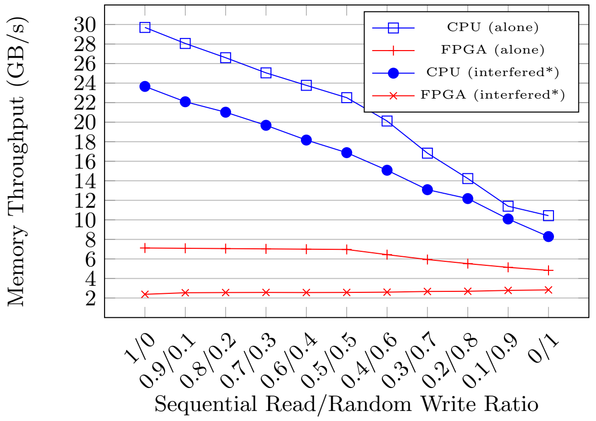


Figure 2: Memory bandwidth available to the CPU and QPI bandwidth available to the FPGA depending on the sequential read to random write ratio. *The FPGA sends a balanced ratio of read/write requests (0.5/0.5) at the highest possible rate, while the CPU read/write ratio changes as shown on the x-axis.

2.2 Effects of the Cache Coherence Protocol

Before discussing performance in later sections, we micro-benchmark the Xeon+FPGA architecture. In the experiments we saw that when the CPU accesses a memory region which was lastly written by the FPGA, the memory access takes significantly longer in comparison to accessing a memory region lastly written by the CPU. In the micro-benchmark, we first allocate 512 MB of memory and either the CPU or the FPGA fills that region with data. Then, the CPU reads the data from that region (1) sequentially, (2) randomly. The results of the single-threaded experiment is presented in Table 1. After the FPGA writes to the memory region, no matter how many times the CPU reads it, it does not get faster. Only after the CPU writes that same region do the reads become just as fast.

This is a side-effect of the cache coherence protocol between two sockets connected by QPI. When the FPGA writes some cache-lines to the memory, the snooping filter on the CPU socket marks those addresses as belonging to the FPGA socket. When the CPU accesses those addresses, they are snooped on the FPGA socket, which causes a delay. Furthermore, the snooping filter gets only updated through writes and not reads. In a homogeneous 2-socket machine with

Table 1: Memory access behavior depending on which socket has lastly written to the memory

	CPU reads sequentially	CPU reads randomly
CPU writes	0.1381 s	1.1537 s
FPGA writes	0.1533 s	2.4876 s

2 CPUs, this is not an issue because both sockets would have the same amount of L3 cache (in this case 25 MB). The probability of a cache-line residing in the L3 of the other socket would be very high, if that cache-line was last written by that socket. However in the Xeon+FPGA architecture, the FPGA has a cache of only 128 KB and any cache-line that is snooped on the FPGA socket is most likely not found. In short, the snooping mechanism designed for a homogeneous multi-socket architecture causes problems in a heterogeneous multi-socket architecture.

This behavior particularly affects the hybrid join because during the build+probe phase the CPU reads regions of memory last written by the FPGA, when it created the partitions. During the build phase the effect is not as high, because the partitions of the build relation are read sequentially. However, during the probe phase, the build relation needs to be accessed randomly, following the bucket chaining method from [21] and the CPU cannot prefetch data to hide the effects of the needless snooping.

In our experiments, the build+probe phase after the FPGA partitioning is always disadvantaged by this behavior. However, the Xeon+FPGA platform is an experimental prototype, and we expect future version not to have this issue.

3. CPU-BASED PARTITIONING

3.1 Background

In a database, a data partitioner reads a relation and puts tuples in their respective partitions depending on some attribute of the input key. This attribute can be determined by either some simple calculation, e.g., taking a certain number of least significant bits of the key in the case of radix partitioning, or something more complex, e.g., computing a hash value of the key in the case of hash partitioning. Thus, the means of determining which partition a tuple belongs to is a factor affecting the partitioning throughput.

Code 1: Partitioning

```
foreach tuple t in relation R
  i = partitioning_attribute(t.key)
  partitions[i][counts[i]] = t
  counts[i]++
```

Another important aspect is how the algorithm is designed to access the memory when putting the tuples into their respective partitions, called the shuffling phase. Since the shuffling is very heavy on random-access, the performance is limited by TLB misses. Earlier work by Manegol et al. [21] has focused on dividing the partitioning into multiple stages with the goal of limiting the shuffling fan-out of each stage, so that TLB misses can be minimized. Surprisingly, the multiple passes over the data required by this approach pay off in terms of performance. Later on, a more sophisticated solution proposed by Balkesen et al. [3] used software-managed cache-resident buffers, an idea first introduced for radix sort by Satish et al. [30], to improve radix partitioning. The cache-resident buffers, each usually having the size of a cache line, are used to accumulate a certain number of tuples, depending on the tuple size. If a buffer for a certain partition gets full, it is written to the memory:

Code 2: Partitioning with software-managed buffers

```
foreach tuple t in relation R
  i = partitioning_attribute(t.key)
  buffers[i][counts[i] mod N] = t
  counts[i]++
  if (counts[i] mod N == 0)
    copy buffers[i] to partitions[i]
```

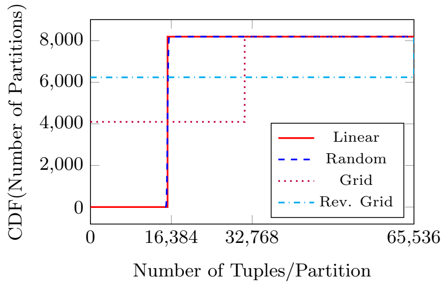
The size of each buffer (N) should be set so that all the buffers fit into L1 to achieve maximum performance. The advantage of this technique is that it prevents frequent TLB misses without the need of reducing the partitioning fan-out. Thus, a single pass partitioning can be performed very fast. An additional improvement to this was proposed by Wassenberg et al. [38] to use non-temporal SIMD instructions for directly writing the buffers to their destinations in the memory, bypassing the caches. That way the corresponding cache-lines do not need to be fetched and the pollution of caches is avoided. If non-temporal writes are used, N depends also on the SIMD width.

Polychroniou et al. [27] and Schuhknecht et al. [32] have both performed extensive experimental analysis on data partitioning and confirmed that best throughput can be achieved with the above mentioned optimizations. Accordingly, in the rest of the paper we use the open-sourced implementation from Balkesen et al. [3] as the software baseline and use a *single-pass partitioning* with *software-managed buffers* and *non-temporal writes* enabled.

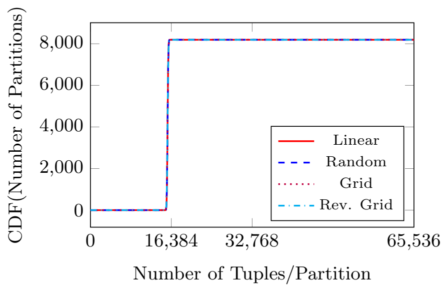
3.2 Radix vs Hash Partitioning

Richter et al. [29] describe the importance of implementing robust hash functions for analytical workloads. In particular, they have shown that for certain key distributions simple and inexpensive radix-bit based hashing can be very ineffective in achieving a well distributed hash value space. On the other hand, robust hash functions can produce infrequently colliding and well distributed hash values for every key distribution but they are computationally costly. Consequently, there is a trade-off between hashing robustness and performance. To observe this trade-off in the context of data partitioning, we perform both radix and hash partitioning on 4 different key distributions (following [29]):

1. Linear Distribution: The keys are unique in the range [1:N], where N is the number of keys in the data set.
2. Random Distribution: The keys are generated using the C pseudo-random generator in the full 32-bit integer range.
3. Grid Distribution: Every byte of a 4B key takes a value between 1 and 128. The least significant byte is incremented until it reaches 128, then it is reset to 1. This is repeated with the next least significant byte until N unique keys are generated.
4. Reverse Grid Distribution: The keys are generated in a similar fashion as the grid distribution. However, the incrementing starts with the most significant byte. Both grid distributions resemble address patterns and strings.



(a) Radix partitioning is applied.



(b) Hash partitioning is applied.

Figure 3: The distribution of tuples across 8192 partitions represented as a cumulative distribution function.

When radix partitioning is used, the resulting partitions may have unbalanced distributions as depicted in Figure 3a. On the other hand, if hash partitioning is used with a robust hash function, such as murmur hashing [2], the created partitions have approximately the same number of tuples as shown in Figure 3b.

Although hash partitioning is robust against various key distributions, using a powerful hash function lowers the partitioning throughput as depicted in Figure 4. However, as the number of threads is increased, the partitioning process becomes memory bound. Consequently, there are free clock cycles available as the CPU waits on memory requests. These free clock cycles can be used for calculating a more powerful hash function. Thus, the throughput slowdown observed with few threads disappears.

3.3 Partitioned Hash Join

The relational equi-join is a primary component of almost all analytical workloads and constitutes a significant portion of the execution time of a query. Therefore, it has received a lot of attention in terms of how to improve its performance on modern architectures. A recent paper by Schuh et al. [31] provides a detailed analysis of implementations and optimizations for both hash- and sort-based joins published in the last few years [3,4,8,19,20]. The conclusion is that partitioned, hardware-conscious, radix hash-joins have a clear performance advantage over non-partitioned and sort-based joins on modern multi-core architectures for large and non-skewed relations. Hence, in the rest of this paper we evaluate how offloading the partitioning phase to a hardware accelerator affects the performance of radix hash join algorithm.

The partitioned hash join algorithm (also known as *radix join*) operates in several stages:

1. Both relations (R and S) are partitioned using radix partitioning, so that each partition fits into cache. More details about the implementation and various optimizations are discussed in Section 3.1.
2. For each partition, a build and probe phase follows:
   (a) During the build phase, a cache resident hash table is built from a partition of R.
   (b) During the probe phase, the tuples of the corresponding partition of S are scanned and for each one, the hash table is probed to find a match.

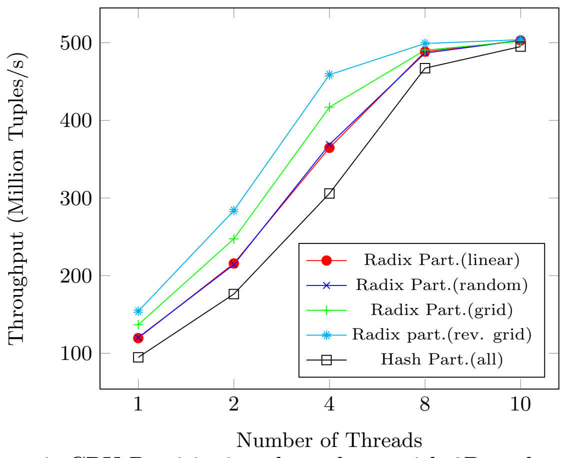


Figure 4: CPU Partitioning throughput with 8B tuples, for varying key distributions and partitioning methods. Hash partitioning delivers for every key distribution the same throughput.

4. FPGA-BASED PARTITIONING

In this section we give a detailed description of the VHDL implementation of the FPGA partitioner. Because the accelerators on the Xeon+FPGA architecture access the memory in 64B cache-line granularity, the partitioner circuit is designed to work on that data width. The design is fully pipelined requiring no internal stalls or locking mechanisms regardless of input type. Therefore, the partitioner circuit is able to consume and produce a cache-line at every clock cycle.

When it comes to tuple width, the design supports various tuple sizes (e.g., 8B, 16B, 32B and 64B). Throughout the first parts of this section, we explain the partitioner architecture in the 8B tuple configuration, since this is the smallest granularity we are able to achieve and poses the largest challenge for write combining. Besides, 8B tuples <4B key, 4B payload> are a common scheme in most of the previous work [4,31] and that allows for direct comparisons afterwards in the evaluation. We later show how to change the design to work on larger tuples and how this considerably reduces resource consumption on the FPGA. Additionally, the partitioner has multiple modes of operation for accommodating both column and row store memory layouts, and for handling data with large amounts of skew, as explained in Section 4.5.

4.1 Hash Function Module

Figure 5 shows the top level design of the circuit. Every tuple in a received cache-line first passes through a hash function module, which can be configured to do either murmur hashing [2] or a radix-bit operation taking N least sig-

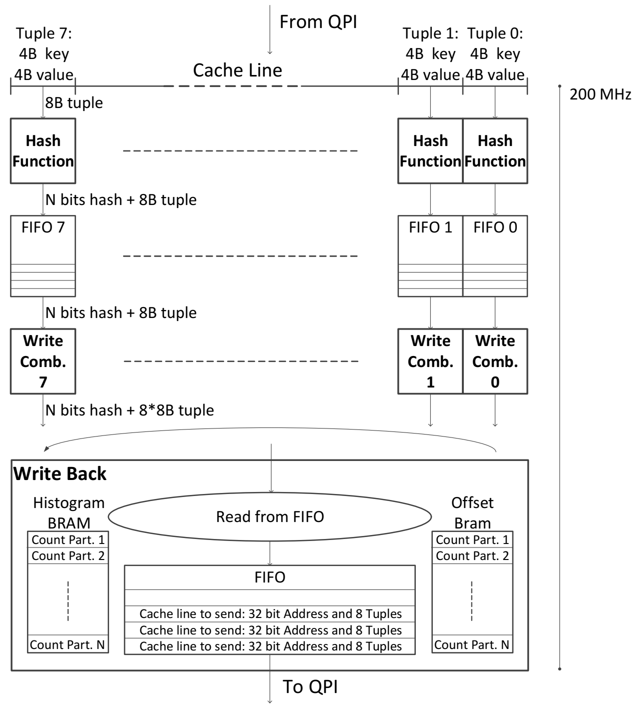


Figure 5: Top level design of the hardware partitioner for 8B tuples.

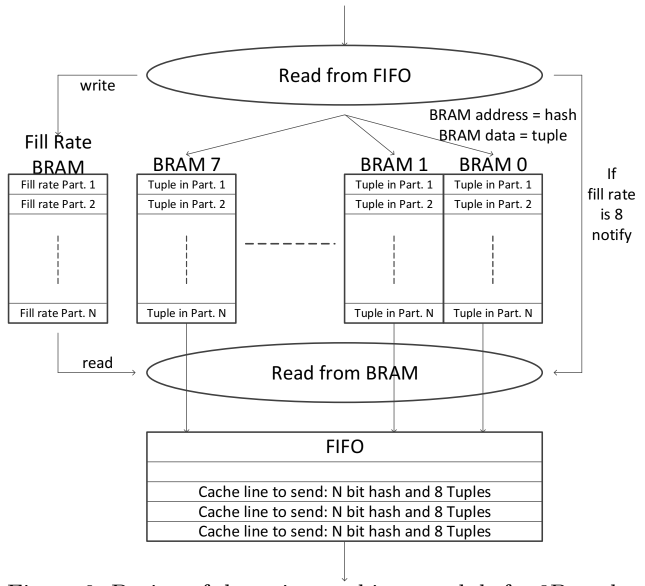


Figure 6: Design of the write combiner module for 8B tuples.

nificant bits of the key. The important thing to keep in mind is that in the synthesized hardware for the pseudo-code in Code 3, every calculation (Lines 6-11) is a stage of a pipeline, which is very different from a sequential execution point-of-view of software. For example, when Line 9 executes the multiplication on the first received key, Line 8 can execute the bit-shift on the second received key and so on. Therefore, the hash function module can produce an output at every clock cycle, regardless of how many intermediate stages are inserted into the calculation pipeline. The only thing that increases with additional pipeline stages is the latency. For murmur hashing the latency is 5 clock cycles: $(1/200MHz) \cdot 5 = 25ns$.

Code 3: Hash Function Module for 4B keys. Pseudo-Code for VHDL, where each line is always active.

```
1  Input is:
2    64-bit tuple <key,payload>
3  Output is:
4    N-bit hash & 64-bit tuple <key,payload>
5  if (do_hash == 1)
6    key ^= key >> 16;
7    key *= 0x85ebca6b;
8    key ^= key >> 13;
9    key *= 0xc2b2ae35;
10   key ^= key >> 16;
11   hash = N LSBs of key;
12 else
13   hash = N LSBs of key;
```

4.2 Write Combiner Module

The output of every hash function module is placed into a first-in, first-out buffer (FIFO), waiting to be read by a write combiner module. The job of the write combiner is to put 8 tuples belonging to the same partition together in a cache-line before they are written back to the memory. To demonstrate the benefits of write combining, consider the case when no write combiner is used: For every tuple going into a partition first its corresponding cache-line from the memory has to be fetched (64B read). Then, the tuple is inserted into the cache-line at a certain offset and written back to memory (64B write). This operation takes place for every tuple entering the partitioner, say T tuples. In total, the amount of memory transfers *excluding* the initial sequential read of tuples is $(64 + 64) \cdot T\, Bytes$. With write combining enabled we have to write approximately as much data as we have read: $64 \cdot T/8\, Bytes$, a 16x improvement over the no write combining option. Since the hardware partitioner in the current architecture is already bound by memory bandwidth, we choose to implement a fully-pipelined write combiner to achieve the best performance on the current platform.

Here we describe in more detail how the circuit avoids internal locking in the write combiner module. The module reads a tuple from its input FIFO as soon as it sees that the FIFO is not empty. It then reads from an internal BRAM the fill rate of the partition, to which the tuple belongs. The fill rate holds a value between 0 and 7 (the number of previously received tuples belonging to the same partition), specifying into which BRAM the tuple should be put (see Figure 6). Reading the fill rate from the BRAM takes 2 clock cycles. However, during this time the pipeline does not need to be stalled since the BRAM can output a value at every clock cycle. The design challenge here is inserting a forwarding register outside the BRAMs, to handle the cases where a read and a write occur to the same address at the same clock cycle. Basically, a comparison logic recognizes if the current tuple goes into the same partition as one of the previous 2 tuples (Lines 6 and 8 in Code 4). If that is the case, the issued read of the fill rate 2 cycles ago is ignored and the in-flight fill-rate of either one of the matching hashes is forwarded (Lines 7 or 9 in Code 4). Otherwise, the requested fill rate is read to determine which BRAM the tuple belongs to. If the fill rate for any partition reaches 7, it is first reset to 0, the tuple is written to the last BRAM, and then a read from the corresponding addresses of all 8 BRAMs is requested. The actual read from the BRAMs happens 1 clock cycle later, since the BRAMs operate with 1 cycle latency. During the read cycle, if a write occurs to the same address as the read, there is no problem because of this 1 clock cycle latency. No data gets lost and no pipeline stalls are incurred regardless of any input pattern.

Code 4: Write Combiner Module. 1d and 2d represent the same signals after 1 and 2 clock cycles, respectively. Pseudo-Code for VHDL, where each line is always active.

```
Input is:
  N-bit hash & 64-bit tuple
Output is:
  N-bit hash & 512-bit combined_tuple

if (hash == hash_1d)
  which_BRAM = which_BRAM_1d + 1;
else if (hash == hash_2d)
  which_BRAM = which_BRAM_2d + 1;
else
  which_BRAM = fill_rate[hash];

if (which_BRAM == 7)
  fill_rate[hash] = 0;
  read_from_BRAM = True;
else
  fill_rate[hash]++;

BRAM[which_BRAM][hash] = tuple;
if (read_from_BRAM_2d == True)
  combined_tuple = BRAM[7][hash_1d] &
                   BRAM[6][hash_1d] & ..
                   BRAM[0][hash_1d];
```

At the end of a partitioning run, some partitions in the write combiner BRAMs may remain empty. In fact, this happens almost every time since the scattering of tuples to 8 write combiners are irregular and some cache-lines remain partially filled in the end. To write back every tuple a flush is performed, where every address of the BRAMs is read sequentially and full cache-lines are put into the output FIFO. To obtain a full cache-line in this case, the empty slots are filled with dummy keys, which later on won't be regarded by the software application. Because of this non-perfect gathering after the scattering, the partitioner circuit writes some more data than it receives. In the worst case, every partition in a write combiner module would be filled with one single tuple, making the other 7 tuples at every partition a dummy key. This overhead is in principle no different than the one incurred through aligning and padding to enable the use of SIMD or optimize cache accesses on a CPU.

4.3 Write Back Module

This module reads the output FIFO of the write combiners in a round-robin fashion and puts the cache-lines in a last stage FIFO to be sent to the main memory via QPI (See Figure 5). There are 2 BRAMs which are used to calculate the end destinations of tuples. The first BRAM holds the prefix sum for the histogram that can optionally be built in an initial run over the data (see Section 4.5). If the histogram is populated, the prefix sum is used to obtain the partition's base address in memory. If the histogram is not populated, a calculated base address via the fixed size partition is used. A second BRAM holds the counts of how many cache-lines have already been written to a certain partition. These counts are used as an offset to the base address to obtain the end address of a current cache-line, which can then be sent. For maintaining the integrity of the offset BRAM, the forwarding logic described in Section 4.2 is used.

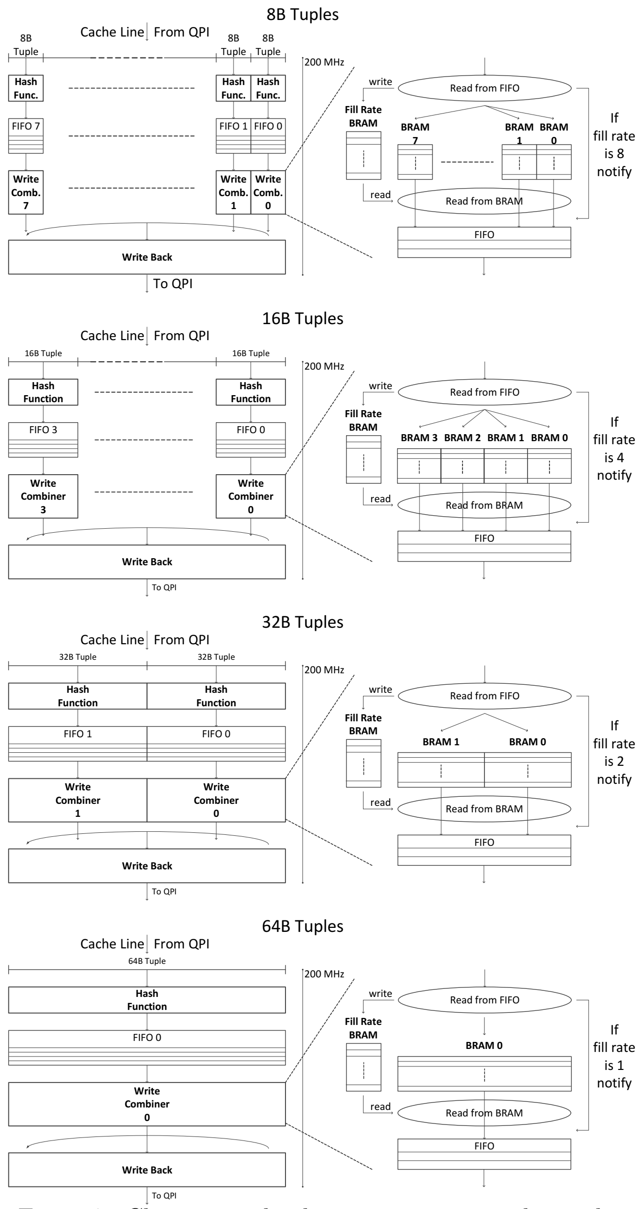


Figure 7: Changes to the design to support wider tuples.

The circuit is able to produce a cache-line at every clock cycle when the entire pipeline is filled, but the QPI bandwidth cannot handle this and puts back-pressure on the write back module. This back-pressure has to be propagated along the pipeline to ensure that no FIFO experiences an overflow, which would cause data to be lost. We handle this by issuing only so many read requests as there are free slots in the first stage FIFOs just after the hash function modules (see Figure 5). If the QPI *write* bandwidth would be more than 12.8 GB/s, which is the output throughput of the circuit, no back-pressure would be put and there would always be free slots in the first stage FIFOs so that new cache-lines would be requested at every clock cycle.

4.4 Configuring for Wider Tuples

The most complex and resource consuming part of the partitioner is the write combiner module, partly since it has to assemble small tuples to build a full cache-line. Therefore, as we increase the tuple width, the complexity of the write combiner decreases considerably and the overall design becomes much simpler as depicted in Figure 7. However, there is also one part of the design that may become more complex with wider tuples: The hash function. With increasing size of the key to be hashed, more arithmetic units of the FPGA or more pipeline stages may be needed. Nevertheless, the throughput remains a cache-line per clock cycle across all configurations. In Table 2 we can observe how the resource usage drops with wider tuples. The only increase observed is for DSP blocks (responsible for arithmetic operations on the FPGA) when going up from 8B to 16B. This is due to the hash function requiring more multiplications per clock cycle to be able to hash 8B keys instead of 4B keys. However, for 32B and 64B tuples, the DSP block usage drops since the write combiner becomes much simpler (less addresses need to be computed).

Figure 8 shows the throughput in tuples per second, which understandably decreases with wider tuples, since the partitioner is bandwidth bound. However, the total amount of data processed remains nearly the same, indicating that the partitioner consumes and produces cache-lines at the same rate regardless of the tuple width (as predicted by the analytical model in Section 4.6).

4.5 Different Modes of Operation

The partitioner has 2 binary configuration parameters, resulting in 4 modes of operation in total.

1. How to format the output: HIST or PAD mode

   (a) Histogram Building Mode (HIST). In this mode the partitioner does a first pass over the relation to build a histogram. During the first pass, no data is written back, and the histogram is built using an internal BRAM. During a second pass, the tuples are written out to their partitions using the prefix sum obtained from the saved histogram. In this mode, intermediate memory for holding the partitions is minimized. This mode is also robust against skew, because the number of tuples in each partition is known before writing out begins.

   (b) Padding Mode (PAD). In this mode the partitioner preassigns a fixed size to every partition, which is calculated by: $\#Tuples/\#Partitions + Padding$. As the padding gets larger, the partitioner becomes more robust against skew. In this mode only one pass over the data is done and the tuples are written directly to their partitions by using the fixed sized prefix sum. If one partition gets filled, the operation aborts and falls back to a CPU based partitioner. This should happen very rarely and only under large skews with a Zipf factor of more than 0.25.

2. Whether the partitioner works in a column store mode or not: RID or VRID

   (a) Record ID Mode (RID). In this mode the tuples reside in the main memory as the partitioner expects them: <xB key, yB payload>.

   (b) Virtual Record ID Mode (VRID) is used by column store databases. In this mode the keys and the payloads are assumed to be stored in separate arrays in the memory, only associated by their ordering in the arrays. However, partitioning does not maintain the ordering of tuples in the intermediate result, that is the created partitions. Therefore, in this mode the FPGA only reads the key array as <xB key> and a virtual record ID is appended to that key on the FPGA, creating a tuple <xB key, 4B VRID>. After the partitioning takes place, the real tuple can be materialized using the VRIDs to associate keys with their payloads.

Table 2: Resource usage depending on tuple width configuration

Tuple width	Logic units	BRAM	DSP blocks
8B	37%	76%	14%
16B	28%	42%	21%
32B	27%	24%	11%
64B	27%	15%	6%

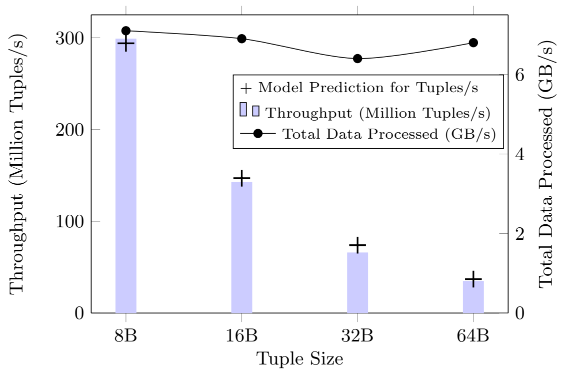


Figure 8: Throughput in tuples per second and total amount of data processed with changing tuple width (Mode: HIST/RID).

4.6 Analytical Model of the FPGA Circuit

Table 3 shows the notation of all parameters that we use in this section when developing the model for the partitioning implementation.

Table 3: Summary of notation used in cost model

Parameter	Description	Values or Units
f_{FPGA}	Clock frequency of the FPGA	200 MHz
T_{FPGA}	Clock period of the FPGA	5 ns
CL	Width of a cache line	64 Bytes
W	Width of a tuple in bytes	8, 16, 32 or 64
r	Seq. read/rand. write ratio	2, 1, 0.5
T_{mem}	Time for memory access	in seconds
$T_{process}$	Time to process	in seconds
B_{FPGA}	Partitioner throughput	in tuples/second
L_{FPGA}	Partitioner latency	in seconds
f_{mode}	Mode factor	2(HIST), 1(PAD)
$B(r)$	QPI Bandwidth for r	in Bytes/seconds
$c_{hashing}$	Cycles for hashing	5
$c_{writecomb}$	Cycles for write combining	65540
c_{fifos}	Cycles for fifo accesses	4

Let's consider the total processing rate of the FPGA partitioner for N tuples and in units of $tuples/s$:

$$P_{total} = min\{\frac{N}{T_{process}}, \frac{N}{T_{mem}}\} \quad (1)$$

The FPGA partitioner completes its execution in:

$$T_{process} = f_{mode}(\frac{N}{B_{FPGA}} + L_{FPGA}) \tag{2}$$

where,

$$B_{FPGA} = \frac{CL}{W} f_{FPGA} \tag{3}$$

$$L_{FPGA} = (c_{hashing} + c_{writecomb} + c_{fifos}) \cdot T_{FPGA} \tag{4}$$

Thus, the processing rate of the FPGA in *tuples/s* is:

$$P_{FPGA} = \frac{N}{T_{process}} = \frac{1}{f_{mode}(\frac{1}{B_{FPGA}} + \frac{L_{FPGA}}{N})} \tag{5}$$

For a sufficiently high N, the term L_{FPGA}/N becomes orders of magnitudes smaller than $1/B_{FPGA}$ and the latency is hidden.

The memory access rate (in *tuples/s*) is, when $r \cdot N$ tuples are read and N tuples are written, where a tuple has W Bytes:

$$P_{mem} = \frac{N}{T_{mem}} = \frac{N}{W(Nr+N)/B(r)} = \frac{B(r)}{W(r+1)} \tag{6}$$

Thus, total processing rate of the FPGA partitioner becomes:

$$P_{total} = min\{\frac{1}{f_{mode}(\frac{1}{B_{FPGA}} + \frac{L_{FPGA}}{N})}, \frac{B(r)}{W(r+1)}\} \tag{7}$$

In the current architecture, the second term in equation 7 is always smaller than the first term. Therefore, it defines the rate at which the FPGA partitioner processes the tuples.

4.7 Performance Analysis

In this section we determine the throughput of the FPGA partitioner using the different modes of operation. Whether radix or hash partitioning is used does not affect performance since computing a hash comes virtually at no additional cost. Figure 9 shows the performance of 4 modes of operation of the FPGA partitioner with respect to prior work, the CPU based partitioning and the raw FPGA partitioning throughput. All experiments are performed on the Xeon+FPGA platform and the numbers represent end-to-end partitioning throughput, with the exception of the raw FPGA numbers. The raw FPGA numbers are obtained with the following method to show the throughput capabilities of the partitioner when it is not limited by the bandwidth: An FPGA internal wrapper around the partitioner is implemented to emulate QPI memory access behavior, however with a combined read and write bandwidth of 25.6 GB/s. The wrapper generates tuples internally, feeds the cache-lines to the partitioner when requested, and gets the processed cache-lines from the partitioner to disregard them.

We observe that using both the PAD and VRID modes increases the throughput. Using PAD instead of HIST is clearly faster, because only one scan over the data has to be done instead of two, although the same amount of data has to be written. Using VRID instead of RID also leads to an increase in throughput. This actually is an experimental proof that the hardware partitioner on the FPGA is memory bound. More specifically, in VRID mode, for each cache-line the FPGA receives, two cache-lines are generated internally by appending the virtual record IDs as explained in Section 4.5. In total the partitioning circuit receives the same number of cache-lines, but over the QPI only half the number of cache lines are read compared to RID mode. Removing some of the reads from the QPI bandwidth lowers the back-pressure on the writes and the overall throughput increases.

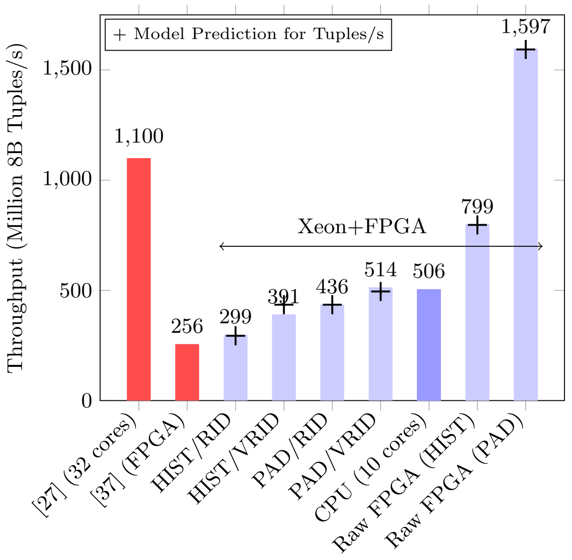


Figure 9: Throughput of hardware partitioner for its 4 different configurations. For all the results the number of partitions is 8192, the tuple size is 8B.

The best direct comparison to CPU partitioning is the HIST/ RID mode, because the CPU algorithm also builds a histogram and uses tuples in <4B key, 4B payload> layout. However, the partitioning algorithm for the CPU builds the histogram out of necessity, in order to remove synchronization between multiple threads, so that each thread accesses a specific part of memory while writing out the partitions. The FPGA implementation can be seen as single-threaded from a software point-of-view because the memory is accessed from only one agent. Therefore, there is no need to build a histogram for synchronization, an advantage for the FPGA partitioner. We see that the FPGA partitioner reaches the same throughput as the 10-threaded CPU partitioner on the Xeon+FPGA platform. When compared to related work, we observe that we improve the FPGA partitioning throughput reported by Wang et al. [37]. In comparison to the results reported by Polychroniou et al. [27], the raw throughput of our FPGA partitioner in PAD mode is 45% higher when compared with 32 cores.

4.8 Model Validation

In this section we validate the model. For this, we select a sufficiently large number of tuples $N = 128 \cdot 10^6$ and assume a tuple width of $W = 8B$:

$$P_{total} = min\{\frac{1 \cdot tuples/s}{f_{mode}(6.25\,\text{e}{-10} + 2.58\,\text{e}{-12})}, \frac{B(r)}{W(r+1)}\} \tag{8}$$

Obviously, the latency term has become 2 orders of magnitude smaller than the output rate for this N, consequently its effect is minimal:

$$P_{total} = min\{f_{mode} \cdot 1.593\,\text{e}9\, tuples/s, \frac{B(r)}{W(r+1)}\} \tag{9}$$

For different modes of operation of the FPGA partitioner, the read to write ratio r changes. From Figure 2 we can look-up the matching bandwidth for a particular ratio $B(r)$ and it should give us the processing rate, which then can be matched to the experimental results in Figure 9.

- In HIST/RID mode the FPGA partitioner reads twice as much data, since the first scan is just for the internal build of the histogram $r = 2$:

$$P_{total} = \frac{7.05GB/s}{8B/tuple \cdot 3} = 294\, Mtuples/s$$

- In HIST/VRID and PAD/RID mode read ratio is equal to write ratio $r = 1$:

$$P_{total} = \frac{6.97GB/s}{8B/tuple \cdot 2} = 435\, Mtuples/s$$

- In PAD/VRID mode read ratio is half the write ratio $r = 0.5$:

$$P_{total} = \frac{5.94GB/s}{8B/tuple \cdot 1.5} = 495\, Mtuples/s$$

Comparing the derived values with the measured ones, we can see that the model matches the experiments within 10%. The model does not capture every detail of the implementation for the sake of simplicity, such as triggering the start of the FPGA partitioning by passing the pointers, the flushing of the pipeline or writing a histogram back. For example, the reason why PAD/RID mode is faster than the HIST/VRID mode in experiments is that, in HIST mode, the FPGA partitioner completely flushes the pipeline while building the histogram during the first phase and then the pipeline has to be filled again during the second phase which adds to the overall latency. We choose not to further detail the model, as the FPGA partitioner remains bound on the memory access bandwidth.

The validated model shows that, if the second term in equation 7 ever becomes larger, by providing a high enough bandwidth around 25.6 GB/s to the FPGA, the first term would define the throughput, which will become 1.6 Billion tuples/s. This is 45% faster than the highest absolute partitioning throughput reported by a 64-threaded CPU solution on a 4-socket 32-core machine [27]. Now, this improvement is achievable on an FPGA with 200 MHz frequency. If the provided design is hardened as a macro on the CPU die, which can then be clocked in the GHz range, one could expect an even higher throughput performance. Even a better utilization of the design would be to have it integrated in a DRAM chip, which could do near memory processing, similar to what Mirzadeh et al. [22] discusses for whole joins.

5. EVALUATION

We evaluate the proposed partitioner when executed as part of a radix hash join. We use the workloads in Table 4 in our experiments, with the key distributions introduced in Section 3.2. All experiments performed here use 8B tuples. Since the join is bandwidth bound both on the CPU and the FPGA, performing it on wider tuples only results in linear changes for tuples per second, whereas total data processed per second remains the same [4].

Table 4: Workloads used in experiments.

Name	#Tuples R	#Tuples S	Key Distribution
Workload A	$128 \cdot 10^6$	$128 \cdot 10^6$	Linear
Workload B	$16 \cdot 2^{20}$	$256 \cdot 2^{20}$	Linear
Workload C	$128 \cdot 10^6$	$128 \cdot 10^6$	Random
Workload D	$128 \cdot 10^6$	$128 \cdot 10^6$	Grid
Workload E	$128 \cdot 10^6$	$128 \cdot 10^6$	Reverse Grid

5.1 Different Number of Partitions

In this experiment we perform the join on workload A with an increasing number of partitions to see how the CPU and FPGA partitioning behave, and how the CPU build+probe phase is affected. Figure 10a shows the results for the single threaded join and Figure 10b for the 10-threaded. When we say 10-threaded join in the context of hybrid joins, we mean that after the FPGA partitioning the CPU build+probe phase is 10-threaded. For the pure CPU join, both partitioning and build+probe phases are 10-threaded. In this experiment, the partitioner can work in PAD mode, since the workload does not have any skew. Also, we choose RID mode so that it is a direct comparison to the CPU.

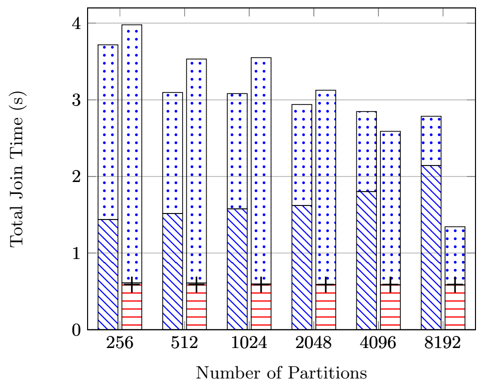
The results indicate that as the number of partitions increases, a single threaded CPU join spends more time on partitioning. On the other hand, FPGA partitioning delivers the same performance regardless of the number of partitions. Similar to the behavior of the CPU join, the build+probe performance increases for the hybrid join as well with increasing number of partitions. For lower number of partitions, the build+probe takes longer time than the partitioning. This is due to the fact that for large relations as in workload A, a low number of partitions is not enough to split the data into small enough, cache-fitting blocks.

The build+probe performance after FPGA partitioning is always slower compared to being performed after CPU partitioning, although both of them execute the same algorithm. This happens due to the cache-coherency protocol as explained in Section 2.2. For the 10-threaded execution depicted in Figure 10b the CPU partitioning becomes slightly faster than the FPGA one. It also appears to be memory bandwidth bound during the partitioning phase, since the performance remains the same across all the number of partitions.

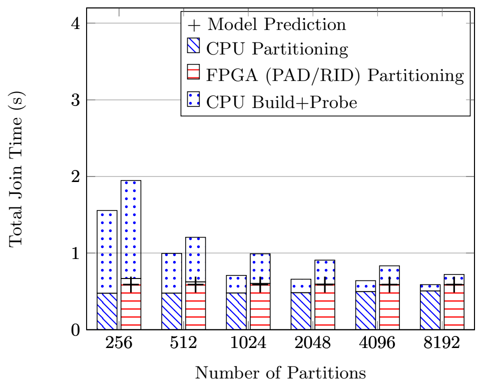
5.2 Different Relation Sizes and Ratios

In this experiment we fix the number of partitions to 8192, which delivers the best performance for build+probe. We perform the pure CPU join and the hybrid join on workloads A and B, representing joins between similar sized relations and joins between a smaller build relation and a larger probe relation, respectively. We observe the join performance with increasing number of CPU threads in Figures 11a and 11b. Again, in the context of the hybrid join the number of threads only refers to the build+probe phase coming after the FPGA partitioning. The partitioner can work in PAD mode, since both workloads are without skew. Otherwise, we choose to evaluate both RID and VRID mode to see how they compare against each other and the CPU.

The FPGA partitioner reaches its best performance in VRID mode, since it has to read half the amount of data from the main memory, saving bandwidth. The hybrid join throughput for workload A in this mode is 406 Million tuples/s and the CPU join throughput is 436 Million tuples/s,

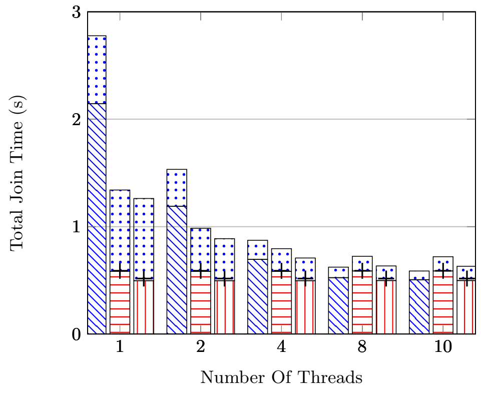


(a) Execution is single threaded.

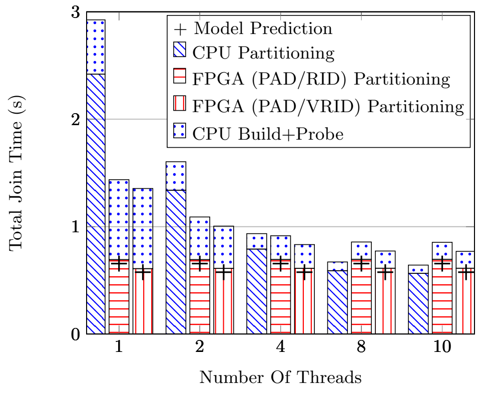


(b) Execution is 10 threaded.

Figure 10: Join performance with increasing number of partitions. Join is performed on workload A.



(a) Workload A



(b) Workload B

Figure 11: Join performance for increasing number of software threads. Number of partitions is set to 8192. Join is performed on workloads A and B.

both reported for 10 threaded execution. In this mode, the FPGA *partitioning* seems to be slightly faster than the 10-threaded CPU one, but the assumption is that the database is a column-store. If the tuples need to be materialized with the payloads later for the query, this will be an additional cost that does not occur in RID mode. However, this is no different than an additional materialization cost that also occurs in column-store database engines.

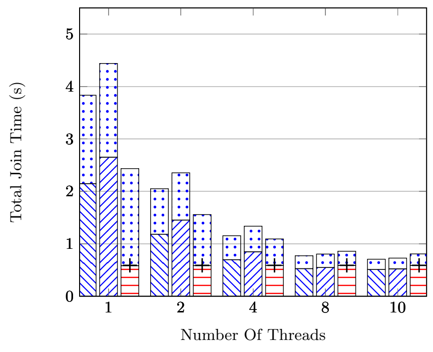
Both CPU and FPGA partitioning for workload B are slightly slower, since the probe relation is larger than the one in workload A and during writing the tuples to their respective partitions random-access over a wider range of memory is required. The build+probe performance after the FPGA partitioning again is throttled by the cache coherence protocol as explained in Section 2.2.

5.3 Different key distributions

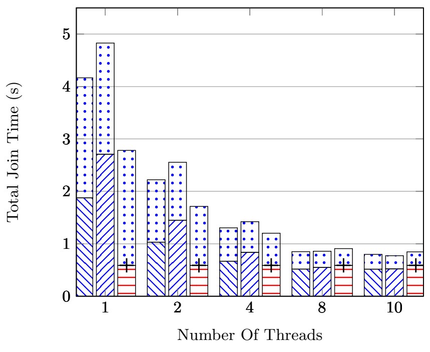
Executing the join for relations of different key distributions provides us a way to see in Figure 12, whether the difference in robustness of hash or radix partitioning has any effect on the performance of the build+probe phase. We perform radix and hash partitioning on the CPU, and just hash partitioning on the FPGA, since hash calculation on the FPGA comes at no additional cost. For workload C, no benefit for the build+probe phase comes from using hash partitioning, since the input keys are distributed randomly. In that case, radix partitioning delivers a good enough distribution. For workloads D and E however, we can observe a visible improvement of build+probe time, when hash partitioning is used: 11% for workload D and 35% for workload E, both observed from the 10-threaded execution. In contrast, the results also show up to 50% increase in the CPU partitioning time when hash partitioning is used, confirming the performance reduction through the added cost of hash computation. For the 10-threaded execution, the CPU does not seem to suffer from doing hash partitioning, because it is memory bound and has free cycles to compute the complex hash function. However, this is only the case for 10-threaded execution, when the whole CPU is used for this operation, whereas with the FPGA we can get the same robustness for free. This is another example where the advantage of the FPGA implementation shows itself, because even a complex hash function calculation does not slow it down and it deterministically delivers the same performance.

5.4 Effect of skew

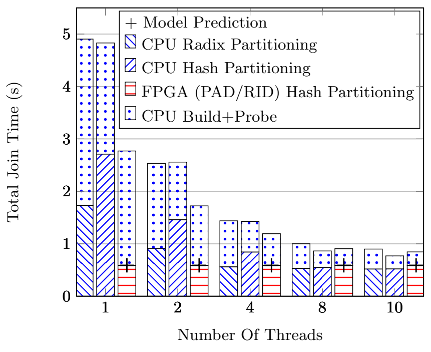
If one of the relations is skewed following the Zipf distribution law with a factor larger than 0.25, the PAD mode of the FPGA partitioner fails for realistic padding sizes, leading to overflowed partitions. When this happens, the HIST mode must be used to ensure no overflow occurring. The detection time for the failure of the PAD mode is random and depends on the arrival order of the tuples, how they are hashed and the size of the relation. The failure is detected



(a) Workload C



(b) Workload D



(c) Workload E

Figure 12: Join performance with for increasing number of software threads. Number of partitions is set to 8192. Join is performed on workloads C,D and E after having either radix or hash partitioning.

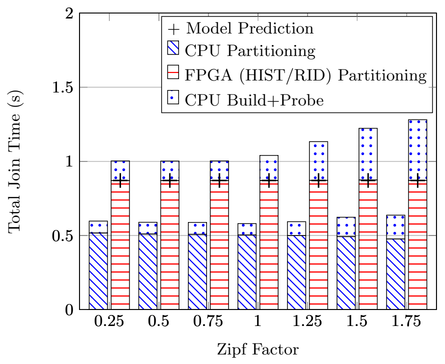


Figure 13: Join performance on workload A, when relation S is skewed. Execution is 10-threaded.

when one of the counters for a partition exceeds the preassigned fixed size. In the worst case, this might happen at the very end of a partitioning run. Then, the procedure has to start from the beginning in HIST mode, which is able handle any Zipf skew factor.

In Figure 13 the join performance for 10-threaded execution is given, comparing CPU partitioning and FPGA partitioning. We see that the FPGA partitioner is slower compared to the CPU one, when HIST/RID mode is used. Note that, this is again a limitation of the bandwidth that is available to the FPGA. As we have shown in our analytical model, the partitioner throughput can reach 800 Million tuples/s in HIST/RID mode, when not limited by the bandwidth (see Section 4.7). This means that the FPGA partitioning time in Figure 13 would be approximately 0.32 seconds, 1.56x faster than the 10-core Xeon.

6. DISCUSSION

This paper's primary focus is on the partitioning sub-operator and its integration within a CPU+FPGA hybrid platform for hash joins, but the proposed techniques and ideas can be applied in a broader setting.

FPGAs excel at processing deep pipelines, vectorized instructions, and computationally intensive workloads. However, their main constraint until now has been the limited integration with the CPU. Therein lies the potential of hybrid architectures. With this work, we have shown that even a tightly integrated and CPU-optimized algorithm as the radix join can benefit from FPGA co-processing. This opens up the possibility of using FPGA co-processing on a wider range of algorithms, from which compute intensive parts can be extracted and accelerated. For example, the partitioning we have described can also be used for a hardware conscious group by aggregation [1] and in other operators involving partitioning [27]. Recent literature provides multiple examples for operations which can be offloaded to an FPGA: Sub-operators such as bitonic sorting networks [10], histograms [15], hash functions [18]; full operators such as pattern matching and regular expressions [14,23]; and eventually new functions typically not supported such as skyline operators [40].

FPGAs are bandwidth bound on most workloads because of their very high internal processing rate and parallelism. Therefore, we currently try to optimize our designs for the limited bandwidth, hence the write combiner in our partitioner. Sequential access (e.g., table scans) and stream processing are something FPGAs are very good at. In the current state where the bandwidth is so scarce, doing random access from the FPGA is not a good option (e.g., using indexes), unless it is part of the algorithm as it was for the partitioner writes. In this context, the data layout is also important to consider. Column stores can benefit greatly from deep pipelines implementing complex analytics queries. Similarly, when processing compressed columns (a de facto standard for analytical workloads), decompression and compression can be done for free on the FPGA as the first and the last steps of a processing pipeline. For row stores, the flexible vectorized instructions (e.g., a customized SIMD) can be very beneficial, enabling multiple predicate evaluations per only one row access.

Regarding the integration of FPGA co-processing into a DBMS, depending on the nature of the algorithm there are several possibilities. Some can be integrated as part of the DBMS query processing by invoking the FPGA sub-operators for a more complex relational operator. This is the method used in this paper. A similar approach was also explored by He et al. in the context of CPU+GPU co-processing [13]. An alternative method for DBMS integration is to implement relational operators or complex analytics as user defined functions (UDF) executed on the FPGA, similar to what Sidler et al. [33] propose.

As part of the future work, we see two main use cases for the partitioner presented in this work. The first one is a tight integration into a DBMS, following the ideas presented in [33], for accelerating end-to-end execution time of joins.

The second one is to have the FPGA partitioner directly connected to the network to distribute the data across machines using RDMA for the highly scalable distributed joins presented by Barthels et al. in [6, 7].

7. RELATED WORK

Relational database joins have been the subject of hardware acceleration in many previous works. In the following, all reported throughput numbers are for 8B tuples and similar relation sizes and ratios to provide as fair a comparison as possible. Kaldeway et al. [17] port hash join algorithms to a GPU exploiting massive parallelism and achieve 480 Million tuples/s throughput. Sukhwani et al. [35] use an FPGA to accelerate predicate evaluation and decompression to improve the performance of queries utilizing those sub-operators. Werner el al. [39] provide one of the early works implementing simple join algorithms on an FPGA at very small scales (evaluation is up to 5000 tuples per relation, reaching a throughput of 2.5 Million tuples/s) and the data is assumed to be on the FPGA prior to the join operation. Halstead et al. [12] implement a non-partitioned hash join on an FPGA and report the simulated throughput for the probe phase, where the FPGA probing outperforms the CPU by an order of magnitude. As the results reported are based on a simulation, data transfer overheads and integration challenges are not addressed. As a follow-up of this work, Halstead et al. [11] implement a non-partitioned hash join on a Convey-MX architecture with multiple FPGAs and shared global memory. The logic on the FPGAs is designed to hide the memory-latency by sustaining a high utilization of the available memory bandwidth (76 GB/s) through deep-pipelining and having thousands of concurrent hardware threads. The design relies on in-order responses to memory requests and direct support of atomic operations for the FPGA, which is currently only available in the Convey-MX architecture. With 4 FPGAs, the system achieves around 620 Million tuples/s join throughput. Casper et al. [9] present a sort-merge based equi-join implemented completely on an FPGA. Based on this work, Chen et al. [10] perform a hybrid sort-merge join and accelerate the sorting phase with a bitonic sorter on an FPGA, which is part of a hybrid CPU-FPGA platform designed mainly for embedded low-end applications. Because of the memory bandwidth limitations of the target platform, the resulting design does not outperform previous work in terms of absolute throughput (with 86 Million tuples/s). Ueda et al. [36] focus more on partial reconfiguration, reconfiguring the FPGA at runtime either with a sort-merge join or a hash join pre-compiled circuit depending on relation sizes to provide optimal performance. Jha et al. [16] and Polychroniou et al. [26] both implement non-partitioned and partitioned hash joins on a many-core (Xeon Phi) architecture. The join throughput reported by Jha et al. [16] is 450 Million tuples/s with the non-partitioned hash join and by Polychroniou et al. [26] 740 Million tuples/s with the partitioned hash join on the Xeon Phi. Another interesting work for accelerating join processing by Mirzadeh et al. [22] suggests doing near memory processing, to execute the join without the CPU ever touching the actual data. Although absolute throughput numbers are not reported, it is stated that near memory join processing outperforms CPU-based ones for both hash and sort-based joins, through a highly parallelized design and very high internal bandwidth. There has been also many advancements for CPU-based join implementations in recent years [4, 5, 8, 20]. Schuh et al. [31] performed a detailed experimental analysis of previous work and suggested their own improvements for NUMA-aware joins. On a server with 4 CPUs each with 15 cores, they report a throughput of 1800 Million tuples/s.

Data partitioning as an important sub-operator in database engines has been studied in previous work. Polychroniou et al. [27] provide an extensive analysis on data partitioning across several dimensions, such as the partitioning type (radix, hash or range) and the shuffling strategy. It has been shown that for more than 16 partitions, write-combining partitioning with non-temporal writes directly to the memory bypassing the cache performs the best. The reported partitioning throughput is 1.1 Billion tuples/s for 8192 partitions with 64-threaded parallel execution on a 32-core server. Schuhknecht et al. [32] present a set of experiments executing radix partitioning. Known optimizations (write-combining, non-temporal stores etc.) are enabled and novel optimizations (prefetching for writes, micro row layouts) are added step-by-step to observe their effects on the total execution time. The experiments in this study are all single-treaded and the reported throughput is 77 Million tuples/s. Wu et al. [41] present a range partitioner accelerator designed as an ASIC and the simulated throughput is reported to be 312 Million tuples/s for 511 partitions. The fastest to date FPGA data partitioning implementation is presented by Wang et al. [37] with 256 Million tuples/s for 8192 partitions. They improved an existing OpenCL implementation of a partitioner and deployed it on an FPGA. The data to be partitioned is assumed to reside in a DRAM connected directly to the FPGA. The resulting partitions are written back to the same DRAM, making a transfer over PCIe to the host memory necessary if the partitioned data is to be used by the CPU for subsequent operations.

For reference, the design we propose in this paper achieves for 8192 partitions a raw partitioning throughput of 1597 Million tuples/s. On the Xeon+FPGA platform, where we prototype our design and perform experiments, we measure a maximum end-to-end partitioning throughput of 514 Million tuples/s. Our hybrid join achieves a maximum throughput of 406 Million tuples/s.

8. CONCLUSION

In this paper we present an FPGA data partitioner for radix hash joins. To our knowledge it is the first FPGA partitioner to avoid internal pipeline stalls and locks, being able to produce a 64B cache-line per clock cycle. We test our implementation on a research-oriented platform, Intel Xeon+FPGA. In an experimental analysis incorporating a wide range of workloads, we show that the hybrid join is on par regarding performance with the state-of-the-art 10-threaded CPU solution. We develop an analytical model of the FPGA partitioner showing that it is bound on the memory bandwidth. We have shown the benefits of using a specialized hardware data partitioner, and hope to integrate it in future platforms with better characteristics.

9. ACKNOWLEDGMENTS

Part of the work of Jana Giceva has been funded by a Fellowship from Google. The Intel HARP v1 platform has been generously donated by Intel.

10. REFERENCES

[1] I. Absalyamov, R. Halstead, P. Budhkar, W. Najjar, et al. Fpga-accelerated group-by aggregation using synchronizing caches. In *ACM DaMoN*, 2016.

[2] A. Appleby. https://github.com/aappleby/smhasher. January 2016.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *IEEE ICDE*, 2013.

[5] R. Barber, G. Lohman, I. Pandis, et al. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[6] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *ACM SIGMOD*, 2015.

[7] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5), 2017.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *ACM SIGMOD*, 2011.

[9] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *ACM FPGA*, 2014.

[10] R. Chen and V. K. Prasanna. Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform. *IEEE FCCM*, 2016.

[11] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. FPGA-based Multithreading for In-Memory Hash Joins. In *CIDR*, 2015.

[12] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, et al. Accelerating join operation for relational databases with FPGAs. In *IEEE FCCM*, 2013.

[13] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *PVLDB*, 8(4):329–340, 2014.

[14] Z. Istvan, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. 2016.

[15] Z. Istvan, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *ACM SIGMOD*, 2014.

[16] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[17] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *ACM DaMoN*, 2012.

[18] K. Kara and G. Alonso. Fast and robust hashing for database operators. In *IEEE FPL*, 2016.

[19] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, et al. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[20] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *In Memory Data Management and Analysis*. 2015.

[21] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, 14(4):709–730, 2002.

[22] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot. Sort vs. hash join revisited for near-memory execution. In *ASBD*, 2015.

[23] R. Mueller, J. Teubner, and G. Alonso. Glacier: a query-to-hardware compiler. In *ACM SIGMOD*, 2010.

[24] N. Oliver et al. A reconfigurable computing system based on a cache-coherent fabric. In *IEEE ReConFig*, 2011.

[25] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *IEEE ICDE*, 2014.

[26] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *ACM SIGMOD*, 2015.

[27] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *ACM SIGMOD*, 2014.

[28] A. Putnam, A. M. Caulfield, E. S. Chung, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ACM/IEEE ISCA*, 2014.

[29] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.

[30] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, et al. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *ACM SIGMOD*, 2010.

[31] S. Schuh, X. Chen, and J. Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *ACM SIGMOD*, 2016.

[32] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. *PVLDB*, 8(9):934–937, 2015.

[33] D. Sidler, Z. Istvan, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *ACM SIGMOD*, 2017.

[34] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[35] B. Sukhwani, H. Min, M. Thoennes, et al. Database analytics acceleration using fpgas. In *ACM PACT*, 2012.

[36] T. Ueda, M. Ito, and M. Ohara. A Dynamically Reconfigurable Equi-Joiner on FPGA. *IBM Tehnical Report RT0969*, 2015.

[37] Z. Wang, B. He, and W. Zhang. A study of data partitioning on OpenCL-based FPGAs. In *IEEE FPL*, 2015.

[38] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par*, 2011.

[39] S. Werner, S. Groppe, V. Linnemann, and T. Pionteck. Hardware-accelerated join processing in large Semantic Web databases with FPGAs. In *IEEE HPCS*, 2013.

[40] L. Woods, G. Alonso, and J. Teubner. Parallel computation of skyline queries. In *IEEE FCCM*, 2013.

[41] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ACM SIGARCH*, 2013.