Bourns College of Engineering
Department of Computer Science and Engineering
351 Winston Chung Hall Riverside,
CA 92521-0144

Technical Report UCR-CSE-2014-MMNNN

FPGA-based Multithreading for In-Memory Hash Joins

Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar and Vassilis J. Tsotras
Deptartment of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
{rhalstea, iabsa001, najjar, tsotras}@cs.ucr.edu

Given the importance of the hash join in query evaluation, many recent efforts have explored ways to best implement hash joins in multicore architectures. Here we present the first implementation for an end-to-end in-memory FPGA hash join. The FPGA uses massive multithreading during the build and probe phases to mask long memory delays while managing all the states locally on FPGA. It uses custom locking bits to enforce synchronization while building the hash table. Throughput results show speedup between 2x and 3.4x over the best multicore approaches on uniform and skewed datasets. However, this advantage diminishes for extremely skewed datasets.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms: FPGA, Processing Engine, Multi-threading

Additional Key Words and Phrases: FPGA, Hash Join, Relational Database

This work has been supported by National Science Foundation awards: CCF-1219180, IIS-1161997 and IIS-1305253.

1. INTRODUCTION

Database analytics help to drive business decisions, and businesses thrive on how quickly and how well they analyze available data. As a result, many companies have emerged in recent years with in-memory data analysis solutions. Oracle's Exadata, and Pivotal Software's Greenplum have built custom machines for memory intensive workloads. FPGA solutions have also emerged through from IBM's Netezza, and Terradata's Kickfire.

The main factor influencing in-memory processing performance is memory bandwidth. Despite the progress made in multi-core architectures, the major performance limitations come from the memory latency (known as the *memory wall*), that restricts the scalability of such memory-bounded algorithms. The most common solution to this problem is building a long hierarchy of cache memories that mask this latency. Cache structures typically occupy over 80% of a modern CPU area.

This paper explores an alternative approach of dealing with long memory latencies by supporting multiple outstanding requests from various independent threads. This multithreading architecture is implemented on FPGAs and is customized to the workload. It is similar to the multithreading approach used in the SUN UltraSparc architecture (for example, the UltraSparc T5 can support eight threads per core and 16 cores per chip [7]). However, because our FPGA implementation is able to support deeper pipelining, it can maintain thousands (instead of tens) of outstanding memory requests and hence drastically increases concurrency.

As an example of our FPGA-based multithreading approach, we implement a hash join operator. Hash-joins are basic building block of relational query processing and various recent works have explored their implementation tailored to multicore CPU architectures [4; 5; 11]. Building a hash table with an FPGA would require massive parallelism to compete with the CPU's order of magnitude faster clock frequency. In turn that means many jobs must be synchronized and managed locally on the FPGA. Building the table on-chip with local BRAMs is another option, as the BRAM's 1-cycle latency removes any need for synchronization. However, current FPGAs only have a few MBs of local BRAMs in total, which limits the build relations to only a few thousand elements [9].

Nevertheless, recent progress in FPGA architecture [1] allows locking of individual memory locations. We leverage this advancement to build the first end-to-end in-memory hash join with an FPGA. Te FPGA masks long memory latencies by managing thousands of threads concurrently without using any caches, opposed to software implementations, which require effective caching to limit memory requests. We analyze and test our design in hardware and show throughput up to 1,600 million tuples per second with 76.8 GB/s memory bandwidth. We also claim 3.4x speedup over the state-of-the-art software implementations when the CPU and FPGA have similar bandwidth.

The rest of this paper is organized as follows: Section 2 discusses related work while our approach is described in Section 3; the experimental results appear in Section 4 and Section 5 provides conclusions.

2. RELATED WORK

Many recent works consider the in-memory implementation of joins (hash or sort-merge). [13] was the first to emphasize the importance of TLB misses in partitioned hash joins and proposed a *radix clustering* algorithm to keep the partitions cache resident. [5] studied the performance of hash joins by comparing simple hardware-oblivious algorithms and *radix clustering* approaches (which are tightly tailored to the underlying hardware architecture). Results showed that the simple implementations surpass hardware-conscious ones. However recently, [4] applied a number of optimizations and found that hardware-conscious solutions in most cases are prevalent over hardware-oblivious.

The implementation of sort-merge joins on modern CPUs was studied in [11], which explored the use of SIMD operations fro sort-merge joins and hypothesized that its performance will surpass the hash join performance, given wider SIMD registers. [2] implemented a NUMA-aware sort-merge algorithm that scaled almost linearly with the number of computing cores. This algorithm did not use any SIMD parallelism, but it was reported to be already faster than its hash join counterparts. Recently, [3] reconsidered the issue and found that hash join still has an edge over sort-merge implementations even with the latest advance in width of SIMD registers and NUMA-aware algorithms.

While the software community has examined both hash and sort-merge joins the FPGA community has concentrated on sort-merge approaches. The reasons for this are twofold. Firstly, sorting and merging implementation is straightforward for parallel FPGA architectures. For example, sorting networks like bitonic-merge [10] and odd-even sort [12] are well established designs for FPGAs; [6] developed a multi-

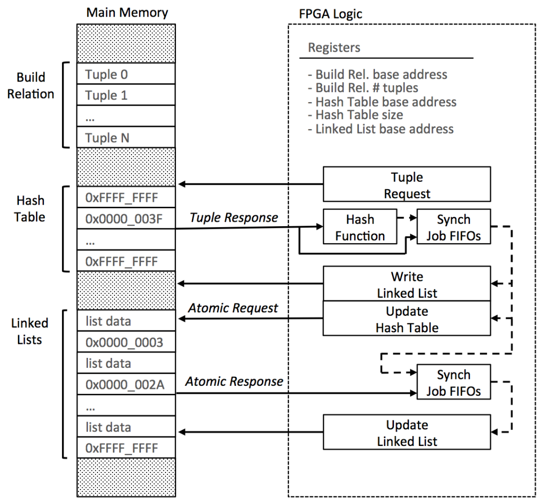


Fig. 1: The FPGA Build Phase Engine.

FPGA sort-merge algorithm, while [14; 18] used sort-merge as part of a hardware database processing system. Secondly, efficiently building an in-memory hash table is non-trivial because of the required synchronization.

3. PROPOSED APPROACH

We outline our implementations for the build phase and probe phase processing engines of the hash join algorithm. We then outline how existing research can be applied to this work and potentially further improve the performance.

3.1 Build Phase Engine

Our target datasets are too large to keep in local FPGA BRAMs. Therefore, our design trades off small and fast on-chip memory for larger and slower off-chip memory. The build engine copes with the long memory latencies by issuing thousands of threads and maintaining their states locally on FGPA. Because of the inherent FPGA parallelism, multiple threads can be activated during the same cycle while other threads are issuing memory requests and going idle.

The entire build relation along with the hash table and the linked lists are stored in main memory (Figure 1). Our hash table uses chaining: all elements hashed to the same bucket are connected in a linked list, and the hash table holds a pointer to the list's head. We use a unique value (0xFFF...FFF) to represent empty buckets in the hash table.

Figure 1 also shows how the build engine (FPGA logic) makes requests to the main memory data structures using 4 channels. In the FPGA logic, local registers are programed at runtime and hold pointers to the relation, hash table, and linked lists. They also hold information about the number of tuples, the tuple sizes, and the join key position in the tuple. Lastly, the registers hold the hash table size, which is used to mask off results from the hash function. The *Tuple Request* component will create a thread for each tuple and issues a request for its join key. The design assumes the join key size is between 1 and 8 bytes, and it is set at runtime with a register. The tuple can be of arbitrary size. If the key is split between two memory locations the *Tuple Request* component will issue both requests, and merge the responses. Requests are continually issued until all tuples have been processed, or the memory architecture stalls. When a thread issues a request the tuple's pointer is added to the thread state, and the thread goes idle.

As join key requests are completed, the thread is activated, and the key along with its hash value are stored in the thread's state. The *Write Linked List* component writes the key and tuple pointer to a new node in the appropriate bucket linked list. The *Update Hash Table* component issues an atomic request to read, and update the hash table. The old bucket head pointer is read while the new node pointer replaces it. An atomic request is needed here because a single engine can have hundreds of threads in flight, and issuing separate reads and writes would create race conditions. While the atomic request is issued the new node pointer is added to the thread's state.

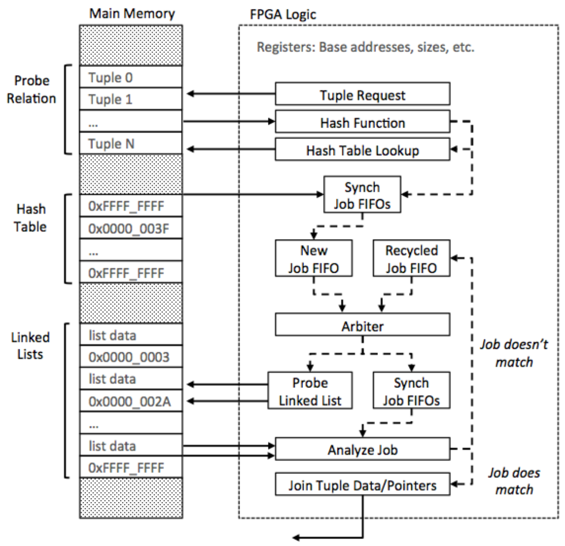


Fig. 2: The FPGA Probe Phase Engine.

As the atomic requests are fulfilled the thread is again activated, and the *Update Linked List* component updates the bucket chain pointer. If no previous nodes hashed to that location then the atomic request will return the empty bucket value, which is used to signify the end of a list chain. Otherwise, the old head pointer is used to extend the list.

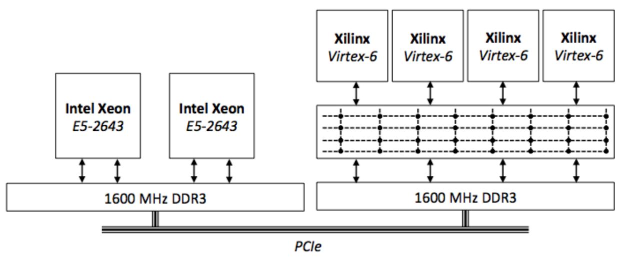
3.2 Probe Phase Engine

The probe engine also assumes that all data structures are stored in main memory. Like the build engine it has to use memory masking to cope with high memory latency and maintain peak performance. Because no data is stored locally, the same FPGA used for the build engine can be reprogrammed with the probe engine; this can be useful in the case of a small FPGA. Larger FPGAs can hold both engines and switch state depending on the required computation.

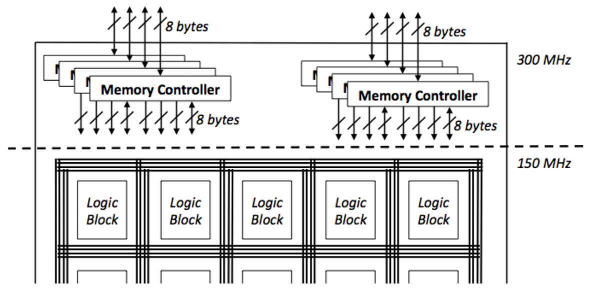
Figure 2 shows how the probe engine makes requests to the data structures in main memory (using 4 channels). Issuing threads, tuple requests, and hashing are handled the same way as in build engine. Again, the join key and the tuple's pointer are stored in the thread's state. Because the probe phase only reads data structures, there is no need for atomic operations. The thread only looks up the proper head pointer by hashed value from the table. The value (0xFFF...FFF) is again used to identify empty table buckets; if this value is returned then the probe tuple cannot have a match and is dropped from the FPGA datapath. Otherwise, the thread is sent to the *New Job FIFO*.

During the probe phase each node in a bucket chain must be checked for matches. A thread is not aware of the bucket chain length without iterating through through the whole chain. Therefore, threads are recycled through the datapath until they reach the last node in the chain. The *Probe Linked List* component takes an active thread and requests its list node. We devote two channels to this component because it issues the bulk of read requests, and its performance is vital to the engine's throughput.

After the node is returned from memory the *Analyze Job* component determines if there was a match. Matching tuples are sent to the *Join Tuple* component. If a node is the last in the bucket chain then its thread is dropped from the datapath. Otherwise, its next node pointer is updated in the thread's state and is sent to the *Recycled Job FIFO*. The datapath can be improved to drop threads once a match is found, but this is only possible if the build relation's join key is unique. An *Arbiter* component is used to decide the next active thread, which will be sent to the *Probe Linked List* component. Priority is given to the recycled threads, thus reducing the number of concurrent jobs and ensuring that the design will not deadlock. Otherwise, when the recycled job FIFO fills, its back pressure would stall the memory responses, causing the memory requests to stall, thus preventing the arbiter from issuing a new job. As matches are found, the *Join Tuple* component merges the probe tuple's pointer (from the thread) with the build tuple's



(a) The Convey MX software and hardware regions.



(b) Convey MX FPGA AE wrapper.

Fig. 3: The Convey MX architecture is divided into software and hardware regions as shown in (a). Each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in (b).

pointer (from the node list) and sends the result out of the engine.

3.3 Possible Optimizations

In practical workloads, joins are typically combined with selections and projections, in an effort to minimize intermediate result sizes (e.g., push selections and projections close to the relation). This approach can also be used here to further improve performance.

Predicate evaluation could filter out tuples, and alleviate memory utilization by creating gaps in the FPGA datapath. This could improve the build phase performance because it removes some of the costly atomic operations. The gaps could also mitigate back-pressure in the probe phase caused by long node chains. By adding the selection hardware on the FPGA, the latency will increase but because it is fully pipelined [16] it would not decrease the throughput.

Projection and the join step (i.e., using the tuple pointers to actually create the joined result) are ideal candidates for FPGA acceleration. Both are naturally parallel and streamable. Many works have leveraged these operations to improve performance [17; 14; 9]. In the special case where an entire tuple fits in one memory word the probe engine presented in this work can be easily extended to perform the join step. The engine already joins the pointers, but a little modification can replace them with the values instead. In order to capture the real effect of FPGA multithreading in the join operation, our implementation does not consider the selection, projection and join step.

Another common optimization applied to multicore hash joins is partitioning, which eliminates the costly thread synchronization and allows to keep partitioned tuples cache resident [15]. However, this idea is not applicable to our FPGA engines. Even with partitioned data, each engine still has hundreds of outstanding read and write requests. Since all these requests are processed in a pipelined manner the only way to avoid race conditions will be to use atomic operations or some form of locking. Moreover our approach does not cache results on FPGA BRAM, hence decreasing the partition size will not have any effect on FPGA performance.

4. EXPERIMENTAL RESULTS

We proceed with a description of the target architecture, the Convey-MX, and discuss how engines can be duplicated to match the available memory bandwidth. The FPGA hash join implementation is compared in terms of overall throughput against the best multicore approach [4]. We also present experiments on the scalability of the FPGA designs and their space utilization. Synthesizing FPGAs is well known to be a time intensive task; nevertheless, all designs in this paper are capable of processing different join queries **without** needing to re-synthesize the FPGA logic.

4.1 Convey-MX Platform

The Convey-MX is a heterogeneous platform with a global memory shared between the CPUs and the FPGAs, allowing us to directly compare hardware and software in-memory hash join applications on the same memory architecture. Figure 3a depicts the MX's memory architecture. It has two regions (the software and hardware) connected though a PCIe bus. Each processor (CPU or FPGA) can access data from both regions, but data accesses across PCIe are significantly longer.

The software region has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache. Multi-socket architecture treats each processor, and the memory, attached to it, as a separate NUMA node with an asymmetry coefficient of 2.0. The software region memory has 128 GB of 1600 MHz DDR3 RAM. Each NUMA node have a peak memory bandwidth of 51.2 GB/s.

The hardware region has 4 Xilinx Virtex-6 760 FPGAs connected to the global memory through a full crossbar. Each FPGA has 8 64-bit memory controllers running at 300 MHz (Figure 3b). The FPGA logic cells run in a separate 150 MHz clock domain to ease timing and are connected to the memory controllers through 16 channels. The hardware region has 64 GB of 1600 MHz DDR3 RAM. Each FPGA has a peak memory bandwidth of 19.2 GB/s. The MX memory also has locking bits at every word block allowing the FPGA to handle synchronization and atomic operations.

4.2 FPGA & Software Implementations

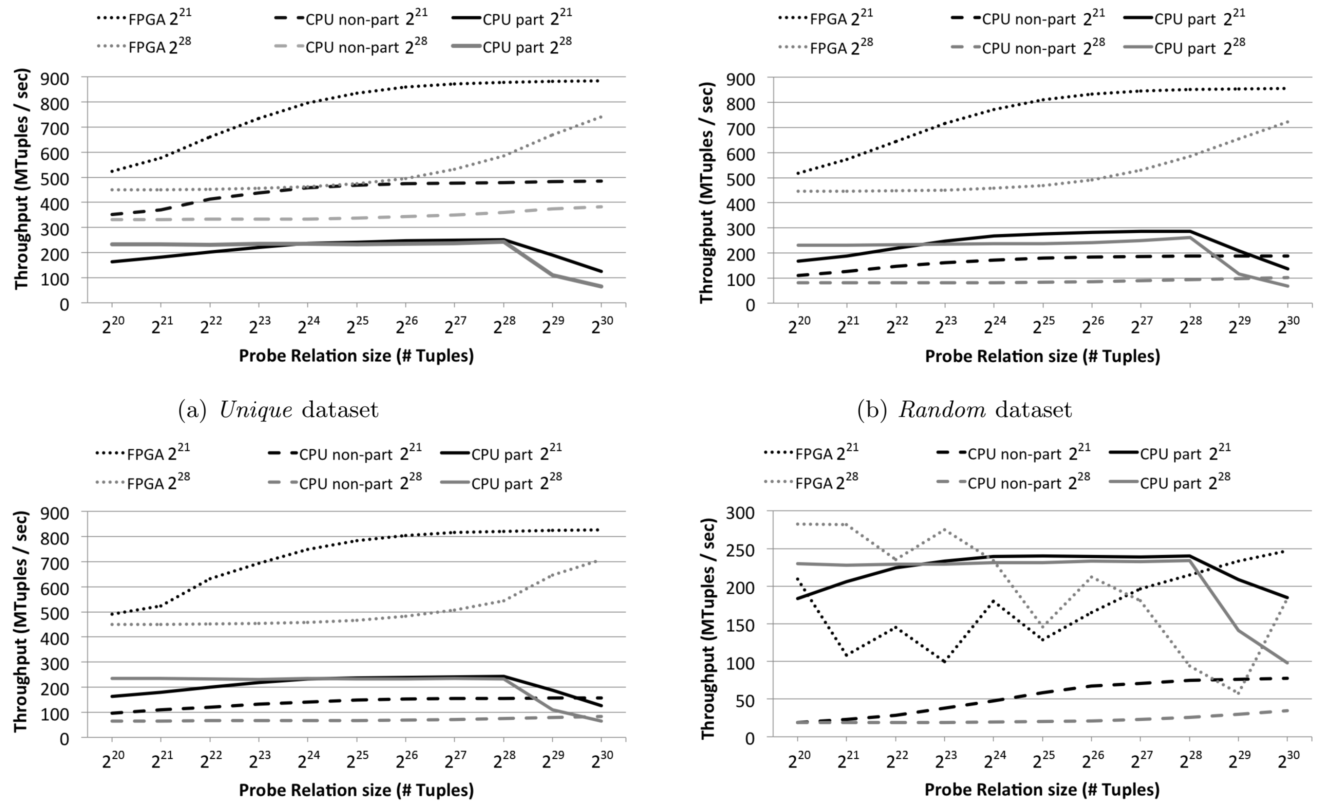
Each FPGA has 16 individual memory channels which is more than what a single build or probe engine would need. To fully utilize the available bandwidth and increase parallelism, we duplicate the number of engines per FPGA. Since the build engine requires four channels (Section 3.1), four build engines can be stored on a single FPGA. Given that each FPGA runs at 150 MHz and, assuming no stalls, one could achieve a peak throughput of 600 MTuples/s per FPGA for the build phase. Similarly, the probe engine (Section 3.2) requires 4 channels, but also jointly uses a channel to write the output result to memory. Therefore only 3 probe engines could be placed on a FPGA. Assuming no stalling the FPGA can reach a peak throughput of 450 MTuples/s per FPGA for the probe phase.

As the state-of-the-art multicore hash join approach we use the implementation from [4]. It includes 2 types of hash join algorithms: a hardware-oblivious non-partitioning join and a hardware-conscious algorithm, which performs preliminary partitioning of its input. Both implementations perform traditional hash join with build and probe phases, however they differ in the way multithreading is used. The non-partitioning approach performs the join using the hash table which is shared among all threads, therefore relying on hyperthreading to mask cache miss and thread synchronization latencies. The partitioning-based algorithm performs preliminary partitioning of the input data to avoid contention among executing threads. Later during the join operation each thread will process a single partition without explicit synchronization. The *Radix clustering* algorithm, which is a backbone of the partitioning stage needs to be parameterized with the number of TLB entries and caches sizes, making the approach hardware conscious. In our experiments we use a two pass clustering and produce 2^{14} partitions, which yields the best cache residency for our CPU.

4.3 Dataset Description

Our experimental evaluation uses four datasets. Within each dataset we have a collection of build and probe relations ranging in size from 2^{20} to 2^{30} elements. Each dataset uses the same 8-byte wide tuple format. The first 4 bytes hold the join key, while the rest is reserved for the tuple's payload. Since we are only interested in finding matches (rather than joining large tuples), our payload is an arbitrary 4-byte value. However, it could just as easily be a pointer to an arbitrarily long record.

The first dataset, termed *Unique*, uses incrementally increasing keys which are randomly shuffled. It represents the case where the build relation has no duplicates, thus keys in the hash table are uniformly distributed with exactly one key per bucket (the all lists are of a size 1). The next dataset (*Random*) uses random data drawn uniformly from an int32 range. Keys are duplicated in less than 5% of the cases for all build relations below 256M. The largest relations have no more than 20% duplicates. For this dataset, bucket lists average 1.6 nodes when the hash table size matches the relation size, and 1.3 nodes when the hash table size is double the relation size. The longest node chains have about 10 elements regardless of the hash table size. To explore the performance on non-uniform input, the keys in the final two datasets are drawn from a Zipf distribution with coefficients 0.5 and 1.0 (*Zipf_0.5* and *Zipf_1.0* respectively); these datasets are generated using the algorithms described in [8]. In *Zipf_0.5* 44% of the keys are duplicated in the build relation. The bucket list chains have on average 1.8 keys regardless of the hash table size, while the largest chains can contain thousands of keys. In *Zipf_1.0* the build relations have between 78% and 85% of duplicates. Their bucket list chains have on average from 4.8 to 6.7 keys. The longest chains range from about 70 thousand keys in the relation with 2^{20} tuples to about 50 million in the 2^{30} relation.



(a) *Unique* dataset

(b) *Random* dataset

(c) *Zipf_0.5* dataset

(d) *Zipf_1.0* dataset

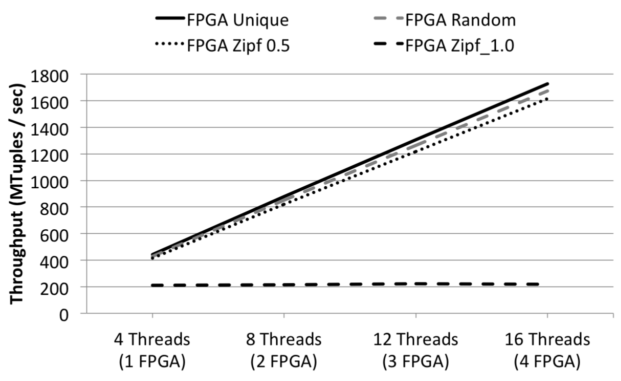
Fig. 4: Dataset throughput as the build relation size is increased.

4.4 Throughput evaluation

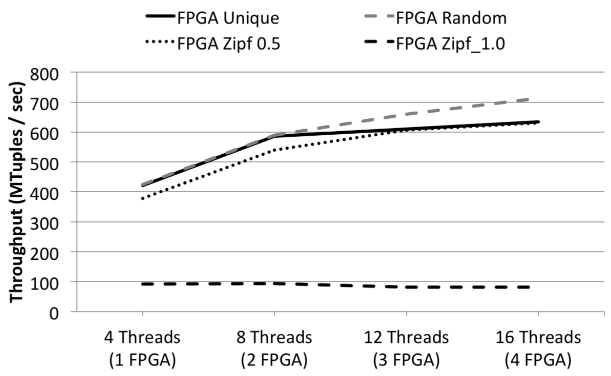
We report the multicore results for both partition-based and non-partitioning algorithms. Results are obtained with a single Intel Xeon E5-2643 CPU. However because of the memory-bounded nature of hash join we use two FPGAs to offset the CPUs bandwidth advantage: a single CPU has 51.2 GB/s of memory bandwidth while two FPGAs have 38.4 GB/s (even with this bandwidth adjustment, the CPU still has almost a 30% advantage).

Figure 4 shows the join throughput for two build relations, with 2^{21} and 2^{28} tuples respectively, while increasing the probe relation size from 2^{20} to 2^{30} for all datasets mentioned in Section 4.3. The FPGA performance shows two plateaus for the *Unique*, *Random* and *Zipf_0.5* data distributions on Figures 4a, 4b and 4c. The FPGA sustains throughput of 850 MTuples/s when the probe phase dominates the computation (that is, when the size of the probe relation is much larger than the size of the build relation) and it is close to the peak theoretical throughput of 1200 MTuples/s which can be achieved with 8 engines on 2 FPGAs. When the build phase dominates the computation atomic operations restrict FPGA throughput to about 450 Mtuples/s (in the FPGA 2^{28} plot, the throughput stays almost constant until the probe relation becomes comparable in size to the build relation). Clearly, in real-world applications the smaller relation should be used as the build relation. In the worst case why we can expect FPGA throughput to be 600 MTuples/s when both relations are of the same size. For the extremely skewed dataset, *Zipf_1.0*, (shown in Figure 4d) the FPGA throughput decreases significantly and varies widely depending on the specific data. This happens because extremely long bucket chains create a lot of stalling during the probe phase that greatly affects throughput.

The CPU results are consistent with those reported in [4]. The partitioned algorithm averages around 250 MTuple/s across all datasets, regardless of whether computation is dominated by the build or the probe phase. It is also not affected by the data skew. For the non-partitioned algorithm, the throughput depends on the relative sizes of the relations, and like the FPGA case, the throughput of the build phase is lower than the

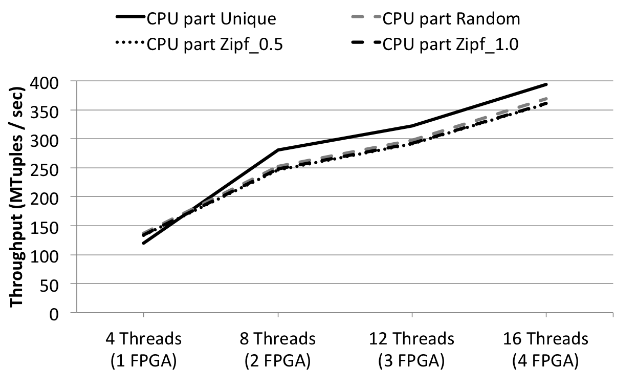


(a) Build Relation has 2^{21}, Probe Relation has 2^{28} tuples

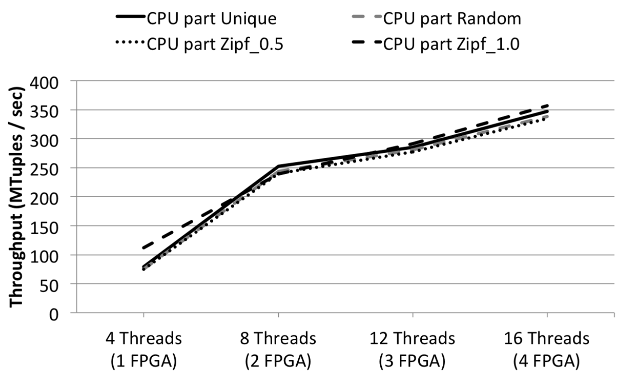


(b) Build and Probe Relation both have 2^{28} tuples

Fig. 5: FPGA Throughput comparison as the bandwidth and number of threads are increased.



(a) Build Relation has 2^{21}, Probe Relation has 2^{28} tuples



(b) Build and Probe Relation both have 2^{28} tuples

Fig. 6: Partitioned CPU throughput comparison as the bandwidth and number of threads are increased.

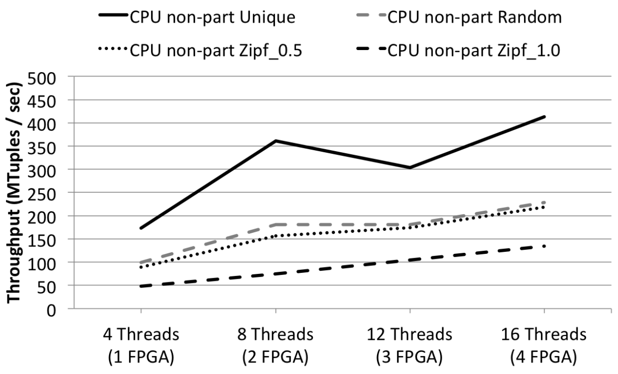
probe phase. The non-partitioned algorithm behaves always worse than the FPGA approach. Interestingly, for the *Unique* dataset, the non-partitioned version has better throughput than the partitioned one, because the bucket chain lengths are exactly one. As the average bucket chain length increases (moving from the *Unique* to the *Random* to the skewed datasets) the throughput of non-partitioned approach decreases. For the extremely skewed *Zipf_1.0* dataset, it falls approximately to 50 MTuples/s.

Averaging the data points within all datasets yields the following results: the FPGA shows a 2x speedup over the best CPU results (non-partitioned) on *Unique* data, and a 3.4x speedup over the best CPU results (partitioned) on *Random* and *Zipf_0.5* data. The FPGA shows a 1.2x slowdown compared to the best CPU results (partitioned) on *Zipf_1.0* data.

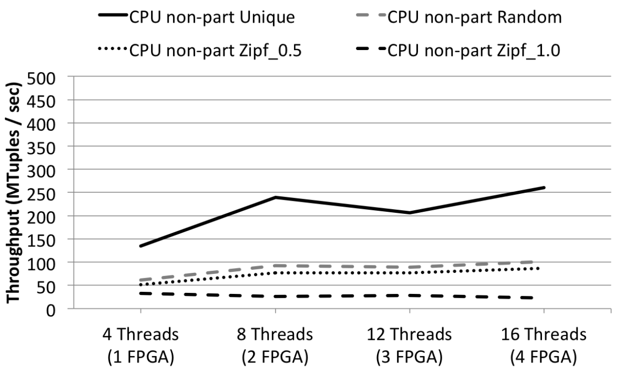
4.5 Scalability

To examine scalability, in the next experiments we attempt to match the bandwidth between software and hardware as closely as possible: every four CPU threads are compared to one FPGA (note that this still provides a slight advantage to the CPU in terms of memory bandwidth). We examine two cases, when the probe relation is much larger than the build one, and when they are of equal size.

Figures 5a,6a and 7a show the results when the probe phase dominates the computation. The FPGA scales linearly on datasets *Unique*, *Random* and *Zipf_0.5* (Figure 5a). However, for the *Zipf_1.0* dataset, the performance does not scale because of the extreme skew. The partitioned algorithm scales as the number of threads increased but at a lower rate than the FPGA approach (depicted on Figure 6a). The non-partitioned algorithm shows a drop in performance while moving from 8 to 12 threads because of the NUMA latency

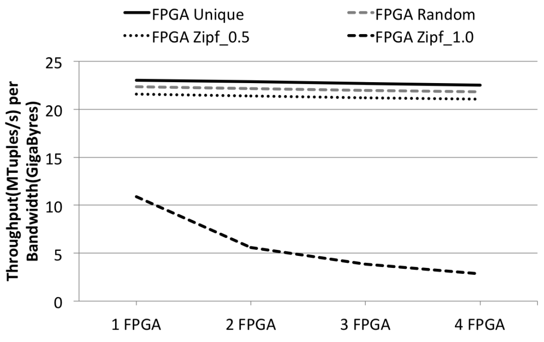


(a) Build Relation has 2^{21}, Probe Relation has 2^{28} tuples

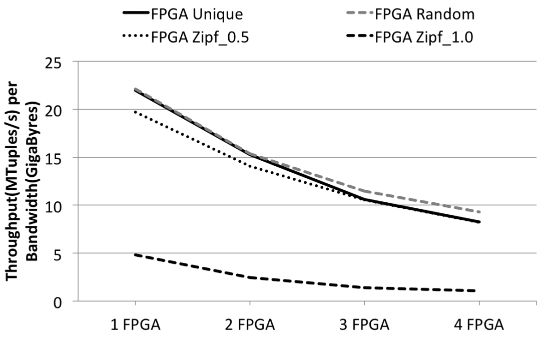


(b) Build and Probe Relation both have 2^{28} tuples

Fig. 7: Non-partitioned CPU throughput comparison as the bandwidth and number of threads are increased.



(a) Built with 2^{21} tuples, Probed with 2^{28} tuples



(b) Built and Probed with 2^{28} tuples

Fig. 8: FPGA throughput normalized to bandwidth.

emerging while moving from 1 to 2 CPUs (Figure 7a).

The FPGA scales at a lower rate when the build and probe relation are of the same size (Figure 5b), since the throughput of the build phase remains constant while the probe phase scales. The slope of the scale graph is almost comparable to the CPU implementations (shown on Figures 6b and 7b) Again the extreme skew case does not scale for the FPGA.

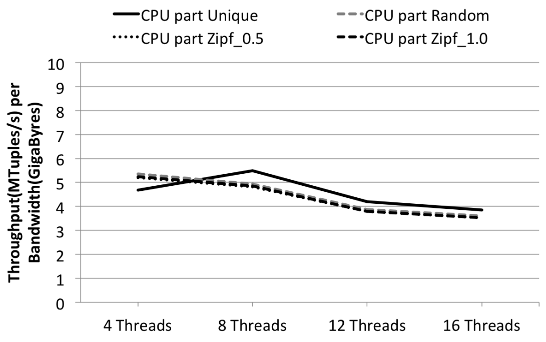
4.6 Normalized Throughput

To get a direct comparison of throughput we normalize it to the bandwidth available. As discussed in Section 4.1 each FPGA has 19.2 GBs of bandwidth, and each CPU has 51.2 GBs. Some of the CPU results only use half the available threads. When doing this each CPU still has 51.2 GBs of bandwidth, but empirically the utilization is halved. Therefore, our normalized results assume a CPU using half the available threads will also have half the available bandwidth.

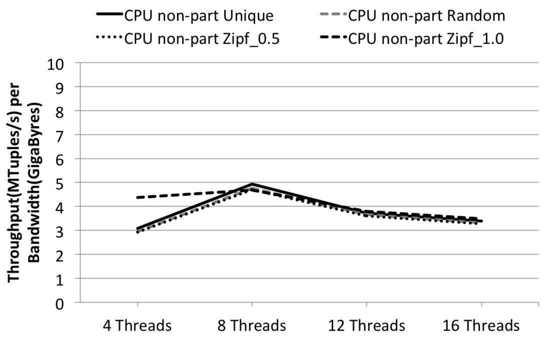
The normalized results are shown in Figures 8, 9 and 10. The FPGA shows speedup of 3.4x to 5.6x on the *Unique* dataset compared to the best CPU results. The FPGA shows speedup of 4.1x to 6.1x on the *Random* and *Zipf_0.5* datasets compared to the best CPU results. Finally, the FPGA shows slowdown of 0.9x to 3.3x on the *Zipf_1.0* dataset. The normalized throughput results are roughly 33% better than non-normalized results from the scalability section (Section 4.5). This is because the CPU has 33% more bandwidth.

4.7 FPGA Area Utilization

Table 1 shows the resource utilization (registers, LUTs and BRAMs used) for the different FPGA designs. We observe that many resources are shared between engines as their number increases. For example, while one probe engine uses 7% of the available registers, whereas three engines utilize only 10% of the register

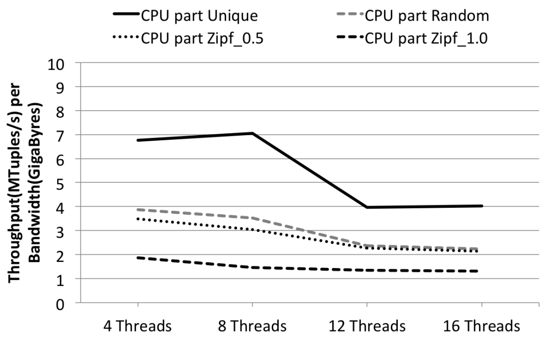


(a) Built with 2^{21} tuples, Probed with 2^{28} tuples

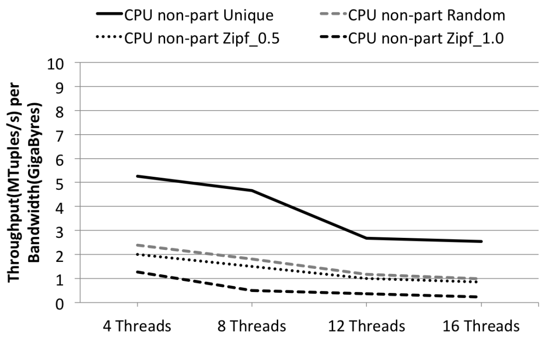


(b) Built and Probed with 2^{28} tuples

Fig. 9: Partitioned CPU throughput normalized to bandwidth.



(a) Built with 2^{21} tuples, Probed with 2^{28} tuples



(b) Built and Probed with 2^{28} tuples

Fig. 10: Non-partitioned CPU throughput normalized to bandwidth.

# Engines	Registers	LUTs	BRAMs
1 probe	65678 (7%)	62521 (13%)	104 (14%)
3 probe	94799 (10%)	86200 (18%)	154 (21%)
1 build	112476 (16%)	118169 (33%)	41 (4%)
4 build	125588 (18%)	135908 (38%)	62 (7%)

Table 1: FPGA Resource utilization.

file. Note that the build phase uses much more logic resources (LUT) due to its atomic operations, but it also has very low BRAM utilization. Overall, the space utilization on the FPGA is low, leaving sufficient space to extend out design for with various optimizations (selections, projections, join step).

5. CONCLUSIONS

We presented the performance benefits of the first end-to-end FPGA implementation of hash joins. Our approach is different as the entire hash-table is built in memory, leveraging FPGA multithreading to deal with long memory latencies. As hashing itself is a basic building block for many relational operator implementations, the presented FPGA design could be extended to support other operations like group-by aggregations, duplicate elimination, unions, intersections etc. Furthermore, we are examining how partitioning and thread load balancing can be utilized on the FPGA approach so as to deal with extremely skewed datasets.

REFERENCES

[1] `http://www.conveycomputer.com/`.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. VLDB'12, 5(10):1064–1075, 2012.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. PVLDB'13, 7(1):85–96, 2013.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. ICDE'13, pages 362–373.

[5] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. SIGMOD'11, pages 37–48, 2011.

[6] J. Casper and K. Olukotun. Hardware acceleration of database operations. FPGA'14, pages 151–160, 2014.

[7] J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols, and A. Vahidsafa. The oracle sparc t5 16-core processor scales to eight sockets. IEEE Micro, 33(2):48–57, 2013.

[8] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly generating billion-record synthetic databases. SIGMOD'94, pages 243–252, 1994.

[9] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with fpgas. FCCM '13, pages 17–20, 2013.

[10] M. Ionescu and K. Schauser. Optimizing parallel bitonic sort. IPPS'97, pages 303–309, 1997.

[11] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. VLDB'09, 2(2):1378–1389, Aug. 2009.

[12] M. Kumar and D. Hirschberg. An efficient implementation of batcher's odd-even merge algorithm and its application in parallel sorting schemes. TC'83, C-32(3):254–264, March 1983.

[13] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. TKDE'02, 14(4):709–730, 2002.

[14] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen. Multi-query stream processing on fpgas. ICDE'12, pages 1229–1232, April 2012.

[15] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. VLDB'94, pages 510–521, 1994.

[16] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using fpgas. PACT '12, pages 411–420, 2012.

[17] J. Teubner and R. Mueller. How soccer players would do stream joins. SIGMOD '11, pages 625–636, 2011.

[18] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. ASPLOS '14, pages 255–268, 2014.