The VLDB Journal (2021) 30:333–359
https://doi.org/10.1007/s00778-020-00642-5

REGULAR PAPER

Efficient local locking for massively multithreaded in-memory hash-based operators

Bashar Romanous[1] · Skyler Windh[1] · Ildar Absalyamov[1] · Prerna Budhkar[1] · Robert Halstead[1] · Walid Najjar[1] · Vassilis Tsotras[1]

Received: 5 February 2020 / Revised: 26 August 2020 / Accepted: 25 October 2020 / Published online: 11 February 2021

Abstract

The join and group-by aggregation are two memory intensive operators that are affecting the performance of relational databases. Hashing is a common approach used to implement both operators. Recent paradigm shifts in multi-core processor architectures have reinvigorated research into how the join and group-by aggregation operators can leverage these advances. However, the poor spatial locality of the hashing approach has hindered performance on multi-core processor architectures which rely on using large cache hierarchies for latency mitigation. Multithreaded architectures can better cope with poor spatial locality by masking memory latency with many outstanding requests. Nevertheless, the number of parallel threads, even in the most advanced multithreaded processors, such as UltraSPARC, is not enough to fully cover the main memory access latency. In this paper, we explore the hardware re-configurability of FPGAs to enable deeper execution pipelines that maintain hundreds (instead of tens) of outstanding memory requests across four FPGAs-drastically increasing concurrency and throughput. We present two end-to-end in-memory accelerators for the join and group-by aggregation operators using FPGAs. Both accelerators use massive multithreading to mask long memory delays of traversing linked-list data structures, while concurrently managing hundreds of thread states across four FPGAs locally. We explore how content addressable memories can be intermixed within our multithreaded designs to act as a *synchronizing cache*, which enforces locks and merges jobs together before they are written to memory. Throughput results for our hash-join operator accelerator show a speedup between 2× and 3.4× over the best multi-core approaches with comparable memory bandwidths on uniform and skewed datasets. The accelerator for the hash-based group-by aggregation operator demonstrates that leveraging CAMs achieves average speedup of 3.3× with a best case of 9.4× in terms of throughput over CPU implementations across five types of data distributions.

Keywords FPGA · CAM · Hash join · Aggregation · Main memory · Relational database

1 Introduction

Fast analytic queries over large collections of customer data are a key workload for any modern business. As the cost of DRAM has continued to decrease, fairly large datasets can now be stored and processed entirely in memory. In recent years, many database vendors have sprung up offering their own in-memory database solutions to speed up processing (MemSQL [44], SQL Server Hekaton [21], SAP HANA [23], Oracle TimesTen [40], IBM DB2 BLU [8]). The main factor influencing in-memory processing performance is memory

✉ Bashar Romanous
broma002@ucr.edu

Skyler Windh
swind001@ucr.edu

Ildar Absalyamov
iabsa001@ucr.edu

Prerna Budhkar
pbudh001@ucr.edu

Robert Halstead
rhalstea@cs.ucr.edu

Walid Najjar
najjar@cs.ucr.edu

Vassilis Tsotras
tsotras@cs.ucr.edu

[1] Department of Computer Science and Engineering, University of California, Riverside, 900 University Ave, Riverside, CA, USA

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. A memory access can stall instruction execution for hundreds of CPU cycles.

The most common solution to the memory latency problem is the use of extensive *cache hierarchies*. This approach mitigates memory latency by relying on data and program instructions localities (spatial and temporal). Unfortunately, such solution does not come for free as cache hierarchies can take up to 80% of a typical processor die area, thus limiting the number of cores that can be accommodated on a single chip and also contributing to energy consumption through leakage current.

Moreover, there are many *irregular applications* that do not exhibit such localities [12]. As a result, cache hierarchies do not provide an effective solution for their memory accesses [14,54]. In particular, irregular applications can be characterized by at least one of the following patterns: (1) Irregular control-flow which breaks the program locality. This is caused by branches in the code that invalidate prefetched instructions. (2) Irregular data-flow where indirection in the memory access patterns breaks the data locality and hence causes cache misses. Some database operators, such as *selection*, exhibit control flow irregularity, while others, like *hash-join* and (hash-based) *group-by aggregation*, can demonstrate both [22].

An alternative for dealing with the memory latency problem in irregular applications is offered by *hardware multithreaded execution* [38,53]. This approach relies on the masking of memory latency by supporting multiple outstanding memory requests and switching to a ready but waiting thread when the currently executing thread encounters a long latency operation, such as a main memory access. Hardware multithreading was used in the SUN UltraSPARC architecture (for example, the UltraSPARC T5 [55]) where it can support eight threads per core and 16 cores per chip. However, tens of threads are not enough to hide memory latency. Instead we have recently advocated for *massive* hardware multithreading implemented on FPGAs (thereafter referred to as multithreaded Processor or MTP) that is able to support deeper pipelining, can maintain hundreds (instead of tens) of outstanding memory requests across four FPGAs and hence can *drastically* increase concurrency and therefore throughput [11,24,28,29].

In this paper, we examine how to use our MTP approach to implement hash-based algorithms for the join [28] and group-by aggregation [1] operators. Both operators are basic building blocks of relational query processor, and various recent works have explored their implementation tailored to multi-core CPU architectures [6,9,16,17,35,59]. A common component in both hash-join and (hash-based) group-by aggregation is to efficiently build a hash table (during the *build phase* of a join or grouping phase of the aggregation), which is later used to join with tuples from the second relation or to return groups in the aggregated relation (potentially with appropriate aggregates). The hash-based nature of these algorithms incurs poor spatial locality, and thus, all the multi-core CPU approaches rely on vast caches to somehow alleviate latency penalty. In order to build a hash table, the MTP execution takes an alternative approach as it requires massive parallelism to compete with the CPU's order of magnitude faster clock frequency. In turn, that means many threads must be synchronized and managed locally on the FPGA. One could instead build a hash table in local on-chip memory (BRAM), as the BRAM's 1-cycle latency removes any need for synchronization. However, current FPGAs only have few MBs of local storage, which limits the hashed relation size only to few thousand records [30].

This paper significantly extends two of our previous works, where we introduced the hash-based join operator accelerator [28] and the hash-based group-by aggregation operator accelerator [1]. Both works are extended by exploring how content addressable memories (CAMs) can be leveraged within the MTP multithreaded designs to enable processor-side locking by acting as a *synchronizing cache*. These CAMs enforce locks and merge threads together before they are written to memory, thus enabling latency-masking of threads [57].

Specifically, in [28] we introduced the first end-to-end hash-join MTP implementation. That design leveraged recent progress in FPGA-based platforms, namely the Convey-MX platform [18], which supports the locking of individual memory locations. The locking bits enable atomic operations that are used to provide synchronization in the build phase. However, the overhead of these atomic operations restricted the practical FPGA throughput to 37.5% of its peak theoretical throughput. One of the major contributions in this work is the elimination of atomic operations and moving all locking on chip by using CAMs for processor-side locking of memory addresses. This results in a more portable, higher performance, and scalable design. Further, the current design is modified to support larger key sizes and arbitrarily wide tuple sizes. This approach is a more practical implementation of relational query processor components rather than a small prototype that works only for fixed sized relations consisting of key and payload. With this extension, we were able to carry further experiments using the TPC-H benchmark. Finally, in addition to inner joins, the updated design also supports an array of join variants: left, right, and full outer join. The above extensions are described in detail in Sects. 3 and 4.

In [1], we applied the MTP multithreading to implement the hash-based group-by aggregation. As group-by aggregation requires updating the hash table (to update the aggregate

as a new record is added to a group), all writes to a hash table need to be synchronized. In [1], we used locks that were implemented at the level of hash table buckets. These coarse-grained locks (CGL) limited parallelism—especially in the case of skewed datasets. Instead, our new design introduces fine-grained locks (FGL), i.e., locks on the record level instead of the hash table buckets, thus reducing locking contention and improving parallelism and throughput. The FGL provides higher parallelism and throughput that is 1.6× to 2.1× times better than the work in [1]. Furthermore, to enable better utilization of memory channels, the new design uses only a single memory channel per engine without any significant effect on throughput. Finally, we implemented both hash-join and hash-based group-by aggregation on a common platform, the Micron (Convey) HC-2ex machine and rerun all experiments on this platform. (These extensions are discussed in detail in Sect. 5.)

The rest of this paper is organized as follows: Sect. 2 gives a description of the target FPGA hardware platforms and background on latency-masking multithreading. Section 3 presents the platform-independent in-memory hash join operator implementation using CAMs. Section 4 describes how the hash-join design can be extended to support larger keys and wide tuples; it also provides an experimental evaluation using the widely adopted TPC-H benchmark. Section 5 describes the implementation of the hash-based group-by aggregation operator and discusses how CAMs can be incorporated in the design. Finally, Sect. 6 discusses related work and Sect. 7 provides conclusions and future work.

2 Background

2.1 FPGA heterogeneous computing

FPGA architectures have evolved from their early stages into large logic arrays capable of concurrently executing multiple complex instructions. They have historically been used as off-chip accelerators where CPUs can offload compute intensive workloads and read back the results. In recent years, the FPGA has been trending closer and closer to the CPU. Xilinx currently offers a Zynq [61] line of chips that couples the FPGA's reconfigurable fabric with an ARM processor, while Intel introduced HARP [51] featuring Xeon+FPGA in a single platform targeting data analysis acceleration [27].

There have been various architectures where FPGA accesses memory directly referred to near-data processing (NDP). Some NDP platforms use FPGA to directly process data stored on none-volatile memory such as the FPGA-Accelerated flash storage proposed in [33] for sorting. Another type of NDP platforms focuses on using the FPGA for data pre-processing, thus reducing performance bottlenecks caused by limited secondary storage and network bandwidth. Examples of such platforms are Netezza [25] used in data analytics and ExtraV framework [41] used in accelerating out-of-memory graph processing. Similar to Netezza, the Aqua platform, used by Amazon Redshift, a distributed and hardware accelerated cache, is another example of NDP technology [4,19]. Nevertheless, it is currently still more common to see the FPGA connected with the CPU over a PCIe bus.

Microsoft Research incorporated multiple Stratix-V FPGAs into a 48 node server that was used to accelerate the Bing search engine [49]. Alpha Data announced a CAPI environment, which allows Xilinx All Programmable devices to connect with IBM Power8 architectures [3]. Multiple companies are offering FPGA platforms over PCIe, which have been actively used in the research community to show acceleration and power savings compared to CPUs and GPUs [13,28].

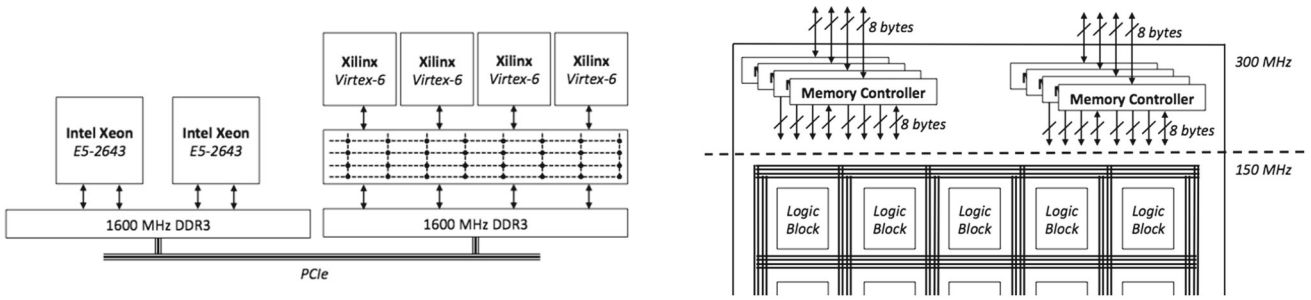
Different FPGA architectures are optimized for various use cases: HPC computations, database workloads, or packet processing. The designs proposed in this paper utilize only the off-chip memory interface, which makes them general enough to be ported between most currently available FPGA platforms. For simplicity, we choose only one platform, the Micron (Convey) HC-2ex, to implement and run all our designs. The Convey architecture offers a shared global memory space between hardware and software, which eliminates any variability due to the memory architecture and allows us to do direct performance comparison against CPU and earlier FPGA implementations.

2.2 Convey heterogeneous computing platforms

The Micron (Convey) HC-2ex is a heterogeneous platform that offers a shared global memory space between the CPU and FPGA regions. As shown in Fig. 1a, the memory is divided into regions connected through PCIe with portions closer to the CPU, and portions closer to the FPGAs. The software region has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache. In total, the software region has 128 GB of 1600 MHz DDR3 memory. The system has a peak memory bandwidth of 51.2 GB/s.

The hardware region has four Xilinx Virtex6-760 FPGAs connected to the global memory through a full crossbar. Each FPGA has 8 64-bit memory controllers running at 300 MHz (Fig. 1b). The FPGA logic cells run in a separate 150 MHz clock domain to ease timing and connect to the memory controllers through 16 channels. These memory channels provide a highly parallel 8,192 simultaneous outstanding requests. The hardware region has 64 GB of 1600 MHz DDR3 RAM. Each FPGA has a peak memory bandwidth of 19.2 GB/s.

The Convey MX, used in our hash-join MTP implementation in [28], has the same hardware layout as the HC-2ex; however, the MX has built-in support for atomic instruc-



(a) The Micron (Convey) HC-2ex software and hardware regions.

(b) Micron (Convey) HC-2ex FPGA AE wrapper.

Fig. 1 The Micron (Convey) HC-2ex architecture. Separation into software and hardware regions in shown in **a**. In the hardware region, each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in **b**

tions directly in memory. The HC-2ex was used for the MTP group-by aggregation implementation in [1]. In this paper, we use the HC-2ex as the common platform for both MTP hash-join and MTP group-by aggregation.

2.3 Latency-masking multithreading

The disparity between main memory latency and processing speed is one of the biggest challenges in computer design. On traditional CPUs and GPUs, it is addressed, by having very large cache hierarchies that mitigate the memory latency by exploiting spatial and temporal data localities. However, large classes of applications have very low levels of data locality and hence do not benefit much from these massive cache hierarchies.

Many database operations fall into this category of applications. The objective of *latency masking multithreading* is to keep the processing unit busy while waiting for memory, thereby masking its latency. For database operations, it is done by fetching and processing as many tuples, one per cycle, as the memory latency in cycles. In our model, we associate a thread with the processing of a tuple. A thread is initiated when a tuple is read from memory and terminated when its processing is completed, whether results are materialized or not. In this paradigm, the processing of an item (a tuple in our case) is halted whenever a memory access is done and that *thread* is put in a *wait state* until the result of the memory access is returned. This means that the *state* of that thread is put in a queue waiting for memory as shown in Fig. 2. When the data are returned from memory, in the case of a memory read, the thread moves to the next stage of its execution and the process is repeated for every memory access that the thread makes until its completion, i.e., until the processing of the tuple is completed. The parallelism, in this model, is equal to the number of *active threads*. A thread is active if it is either *executing* or *waiting*. In this model, a thread goes through an execution pipeline consisting of

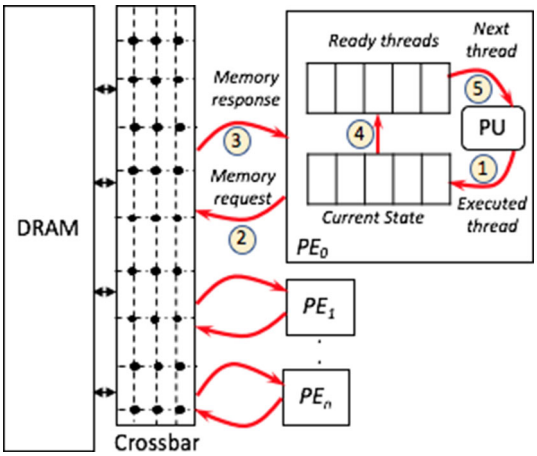


Fig. 2 Components of a multithreaded implementation: Numbers represent execution steps performed by a processing unit (PU) of each processing engine (PE). The design is scaled by adding as many PEs as possible

successions of processing stages and waiting stages. Once this pipeline is full, the execution reaches a steady state and achieves the maximum throughput of one result per engine per cycle and memory latency is fully masked.

Nevertheless, a synchronization mechanism must be provided when threads may write in memory. The main contribution of this paper is a processor-side synchronization mechanism relying on content-addressable memories (CAMs) described in the next section. This mechanism is designed to synchronize all the threads *within* an engine.

On the Micron (Convey) HC-2ex, the average memory latency is $\sim$ 100–200 cycles and memory accesses are fully pipelined, meaning that the processing unit can issue one memory request per cycle per memory channel up to the capacity of the memory buffer, which is $\sim$ 500 memory requests. All memory requests, on the same channel, are returned in the order they were issued. There are three stages in the multithreaded execution that occur in the following

order: (1) fill-up, (2) steady-state, and (3) drain-out. In the fill-up stage, the processing pipeline is being filled. Its duration is determined by the pipeline latency and the service time of each tuple. Conversely, the drain-out stage is when all the tuples have been read from memory and no new ones are inserted in the pipeline. When the number of tuples is much larger than the pipeline latency, the duration of the fill-up and drain-out stages is dwarfed by that of the steady-state, and hence, the throughput is closer to the maximum achievable throughput.

In the experiments described in this paper, we implement a number (N) of engines on each FPGA. The total number of engines is therefore $4N$. Each engine is connected to M memory channels on its FPGA. The data to be processed are partitioned equally among all the engines. All the engines operate concurrently and do not interact.

There is an inherent trade-off between the two parameters N and M, as well as with the allocation of resources on the hardware to implement a given configuration. The choice of these two parameters is discussed in Sect. 3.3 for the MTP hash-join and in Sect. 5 for the MTP group-by aggregation.

The maximum achievable throughput in this model is determined by two factors: (1) the memory bandwidth and (2) the number of active threads per engine. On the HC-2ex, each FPGA has 16 memory channels, and each channel can support one memory operation per cycle at 150 MHz. Therefore, the whole system can achieve a peak of 64 memory operations per cycle, or 9.6 billion memory operations per second. The number of active threads within an engine is limited by the available resources on the FPGA. In an FPGA design, every hardware item must be mapped, by a software tool, on some logic resources on the FPGA and items are connected together by routing wires between them. The length of the longest wire determines the clock frequency of the whole design. A design is said to meet timing if all the wires can be clocked at the target clock frequency. Selecting the best possible routing for every wire is an NP-complete problem. Very good heuristics take hours and sometimes days to achieve a routing that meets timing. The internal architecture of the Xilinx Virtex 6 FPGA is particularly challenging at meeting timing.

CAM-based synchronization is discussed in Sect. 3.3 for the MTP hash-join, and in Sect. 5 for the MTP group-by aggregation. In both cases, the size of the CAM was the main limitation on the number of active threads. The limitation on the CAM size comes from the impossibility of meeting timing for larger CAMs.

2.4 CAMs as synchronizing caches on FPGAs

A CAM (also known as an associative memory) is an array that can perform efficient entry-matching (i.e., answer membership queries). Its operation is the inverse of a random access memory (RAM): When presented with a *search word*, the CAM returns all the locations whose content matches that word. Each CAM bit consists of a flip-flop with a comparator matching it to the corresponding bit in the search word. The outputs of all the bit positions in a word are ANDed to generate the (mis)match for that word. The CAM's ability to perform a search in unit time comes at a high cost of area, energy, and long clock cycle time (due to the long wires for the bit-wise AND and propagating the search word to all the entries). A CAM with n entries where each entry is w bits, stores n words of w bits, by construction. Each entry has a free bit, and the next word to be inserted in the CAM is inserted in the first free entry and when a word is deleted from the CAM that entry is marked free. In this paper, depth or size of a CAM refers to the number of entries in the CAM.

As the number of entries in the CAM increases, the achievable clock frequency of the circuit drops. This limitation either restricts the size of the CAM or increases the number of cycles it takes to perform an update operation. Nonetheless, CAMs have proven to be very useful in domains such as networking (e.g., implementing an IP table in a network router). Recently, we explored how CAMs can be used to accelerate the breadth first search algorithm [57]. These applications can usually tolerate long update latencies because update operations are infrequent.

In a streaming environment, CAMs can maintain a cache of recently seen unique items and allow quick access to them without incurring long ready latency and stalling the pipeline. This fast cache look-up mechanism can also be used as a fine-grained address-based synchronization primitive, which avoids long latency trips to main memory and does not require special hardware synchronization primitives.

Consider the case when a CAM is assigned to guard a particular memory partition. It can be configured to hold the addresses of the values that need synchronized access. If all memory requests within a partition are first submitted to the CAM, before being routed to the memory, the accesses to identical addresses are serialized locally in the CAM. In this case, a CAM entry serves as an exclusive lock, which gets released (flushed from the CAM) after the request(s) completion. In Sect. 5.2, we discuss how to use this approach for synchronization in the MTP group-by aggregation algorithm.

In our previous work [28], we have used atomic operations which were implemented using locks on individual memory locations, provided by the now discontinued Convey MX architecture [18]. Leveraging CAMs for synchronization of FPGA algorithms increases the portability of our design. Locking using generic CAMs means that all synchronization operations are now internal to the FPGA and can be done on any architecture where an FPGA with a sufficient area has direct access to the memory. In addition, this design provides more selective fine-grained synchronization primitives in comparison with the Convey-MX, which places a lock

on all FPGA-memory communication channels. We discuss resource trade-off for CAMs in Sects. 3.1 and 5.3.1. Work on previous FPGA implementations of CAMs appears in Sect. 6. An excellent description of CAMs and their applications can be found in [37].

3 MTP hash join using CAMs

The presented hash join engine in our prior work [28] is in line with previous research efforts [6,9,35], where all relations were broken into "skinny" tables with small tuples (8 or 16 byte key/value pairs). The focus is on finding matching tuples between relations, but we also outlined how the approach can be extended with existing research to further improve performance. The work in [28] relied on atomic instructions provided by the Convey MX machine for synchronizing hash table updates. In this section, we describe how the build phase can be made platform agnostic using CAMs for synchronization and how the probe phase can be extended to support left, right, and full outer joins. We target datasets that are too large to store locally on the FPGA, and therefore, all data structures are stored in global memory. The join phases are nicely split such that every tuple in the build relation incurs write to the data structures, and every tuple in the probe relation only reads from these data structures. This separation simplifies the hash table operations for both the CPU and MTP design, compared to other hash-based algorithms (i.e., group-by aggregation in Sect. 5).

3.1 Build phase engine

Since our target datasets are too large to keep in on-chip memory (e.g., BRAMs), our design trades off small and fast on-chip memory for larger but slower off-chip memory. The build engine copes with the long memory latencies by issuing thousands of threads across all PEs and maintaining their states locally on the FPGA. Because of the MTP design's inherent 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 is stored in main memory as shown in Fig. 3. Our hash table uses the chaining collision resolution technique: 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 special value (0xFFF...FFF) to represent empty buckets and end of chains in the hash table.

Figure 3 also shows how the build engine (FPGA logic) makes requests to the main memory data structures using four channels. Figure 4 shows the state diagram for a single thread of an engine during the build phase. In the FPGA logic, local registers are programmed at runtime and hold pointers

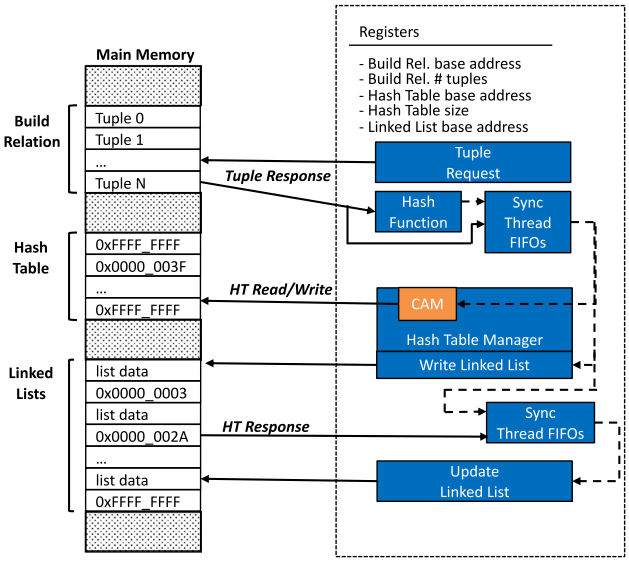


Fig. 3 The MTP build phase engine

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 within 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 4 bytes, and it is set at runtime with a register. In Sect. 4, we show how this design can be extend to handle larger keys and tuples of arbitrary sizes. Requests are continually issued until all tuples have been processed, or the memory architecture issues a stall. Once 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 reactivated and the key and hash value are stored in the thread's state. The *Write Linked List* component writes the key and tuple pointer to a new node into the appropriate linked list bucket. Instead of issuing an atomic swap command as in the previous design [28], the *Hash Table Manager* component issues several memory requests protected by the CAM to do the update. First, the thread searches the CAM for the hash table bucket address. If the address is in the CAM, the thread must wait in a FIFO and wait as it is already being updated. If the address is not found, the thread writes the address to the CAM and now has exclusive access to the hash table bucket. With the lock secured, the thread issues a read for the old bucket head pointer. Once we get a response from memory, the old bucket head pointer is added to the thread's state and the thread issues a write of the updated value. When memory responds with a write complete message, the address is flushed from the CAM, freeing the lock for other threads. Synchronization is needed here because a single MTP engine can have hundreds of threads in flight, and issuing separate

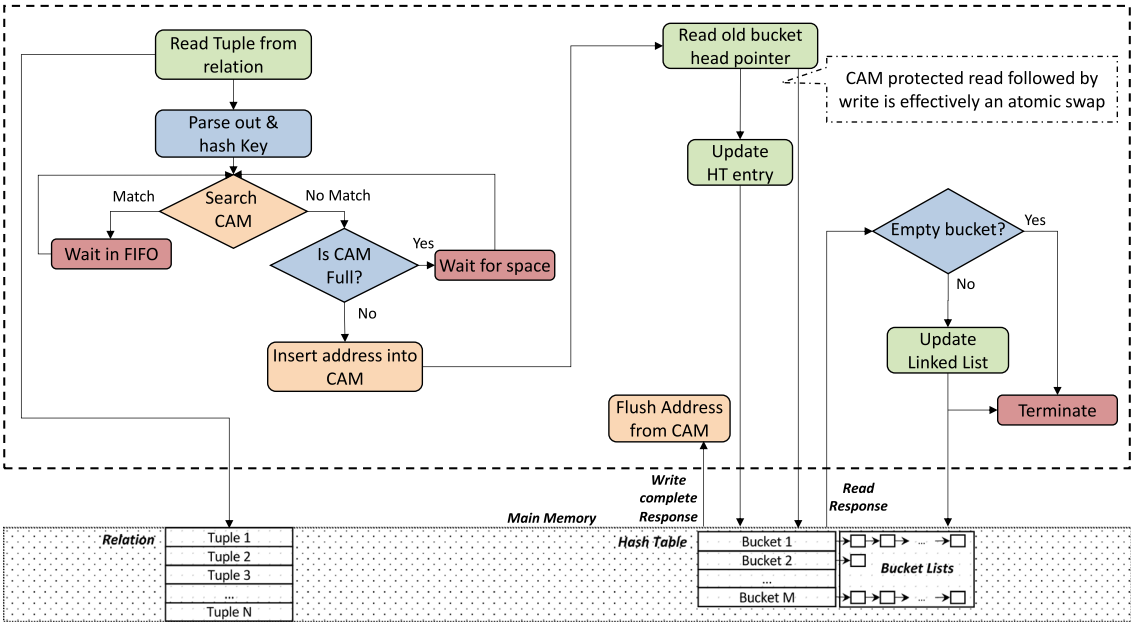


Fig. 4 A state diagram of the steps that every thread goes through in the MTP build phase engine

unprotected reads and writes would create race conditions. After the write is complete, the new node pointer is added to the thread's state.

As the write requests are fulfilled, the thread is again reactivated, and the *Update Linked List* component updates the bucket chain pointer. If no previous nodes hashed to that location, then the hash table read request will return the empty bucket value, which by design is also used to signify the end of a list chain. Otherwise, the old head pointer that was returned is used to extend the list.

The key insight to this design is to realize that all items in the relation must end up in the linked list memory space. Since we know the number of elements, we know exactly how much memory we need in the linked-list node space. Instead of dynamically allocating nodes with keys as we see them, the keys can have a fixed location in memory and it is the next pointer slot that changes based on the current state of the hash table. We are effectively building the linked list around the nodes as they sit in memory. If there are n elements in the relation, the linked-list structure will be $2{*}n$ (one word for the key, and one word for the next pointer). Every $i'th$ element will always end up in the $2 * i$ position and its next pointer will always be at $2 * i + 1$. The dynamic nature of the list building comes from swapping the pointers out of the hash table to write into the linked-list table.

Figure 5 shows an insertion example. The left-side tables show the hash table (HT) and the linked list (LL) after elements k_0, k_1, k_2, k_3 have been inserted, where k_i is a pointer to the key i in main memory. k_0 and k_3 have been hashed to the same bucket in the HT. Therefore, the *next_node* of k_3 in the LL points to address 0 that contains k_0. The right-side

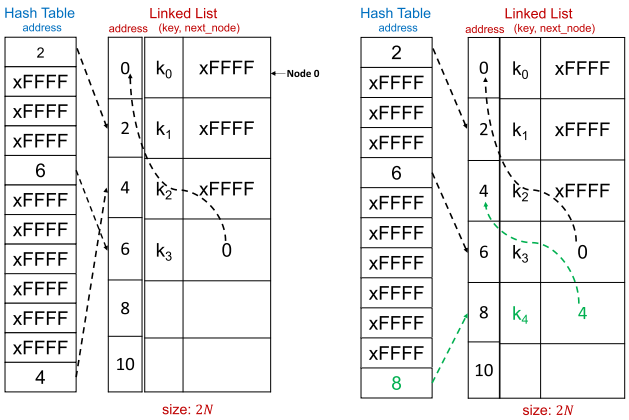


Fig. 5 Example of inserting an element in the hash table. Dashed lines represent logical connections. k_i is a pointer to the key i in main memory. Elements k_2 & k_4 hash to the same bucket in the hash table. k_4 replaces k_2 as the head of the linked list of their bucket

tables show the state of the HT and LL after the insertion of k_4, assuming that the hash function, hashed k_4 to the last bucket in the hash table. k_2 was hashed by a previous thread to the same bucket as well. Hence, the last bucket at the hash table already points to address 4 in the linked list which contains k_2. The new address 8, which contains k_4 in the linked list, will be inserted in the last HT bucket, while the previous address 4 at that bucket will be moved to the *next_node* memory location of k_4 in the LL.

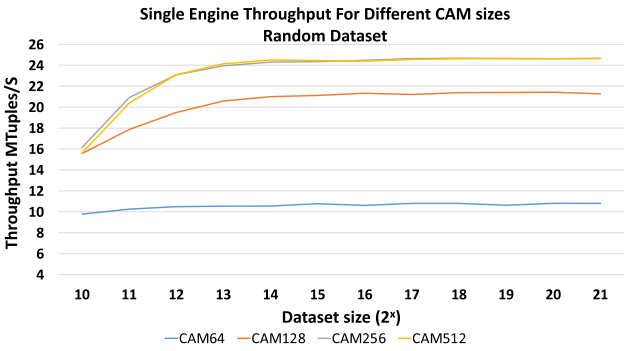


Fig. 6 MTP hash-join build throughput of single engine as the *CAM* size is varied from 64 to 512

Effect of CAM size in MTP hash join The throughput of a MTP engine is proportional to its ability to fully mask memory latency by having multiple concurrent active threads. The maximum number of concurrent threads is set by the size (depth) of the CAM: Each entry in the CAM represents a thread updating the hash table. Figure 6 shows the throughput as a function of the CAM size for uniform distribution for CAM size from 64 to 512 entries. The large increase in throughput, 2.2×, from 64 to 128 entries demonstrates the CAM's ability to act as a cache. However, at 256 entries the throughput is only 1.1× higher than at 128. A CAM size of 512 yields nearly the same performance as that of 256.

Meeting timing (i.e., achieving clock frequency of 150 MHz) on the Xilinx Virtex6 (2012) with one CAM per engine and four engines per FPGA was extremely difficult with a CAM size larger than 128. However, on newer FPGAs, deeper CAMs can be constructed with higher frequencies. The other limit on the depth of the CAM is the maximum number of outstanding memory requests supported by the machine (see Sect. 2.2), which is independent of the FPGA. Thus, the maximum number of concurrent threads is dependent on both: the size of the CAM as well as the supported number of outstanding memory requests.

Our current design uses a CAM size of 128 in order to meet timing requirements without reducing the number of engines.

3.2 Probe phase engine

The probe engine also assumes that all data structures are stored in main memory. Like the build engine, it uses memory masking to cope with high memory latency and maintain peak performance. Because no data are stored locally for either engine, the same FPGA used during the build phase can be reprogrammed with the probe engine (useful for smaller FPGA). Larger FPGAs can hold both engines and switch state depending on the required computation. Our prior design of the probe engine [28] only handled inner-join queries. If key_a and key_b are in both the build and probe tables, then the joined tuple was emitted. In this current design, we support the full array of join variants: inner, left, right, and full outer join. These can be toggled on and off in the data path to configure the join operation desired. Left outer join has no performance overhead as the work to identify the null match is already done while answering an inner join. For the right outer join, the design incurs one final scan of the build table to emit the tuples in the build table that were not matched during the query.

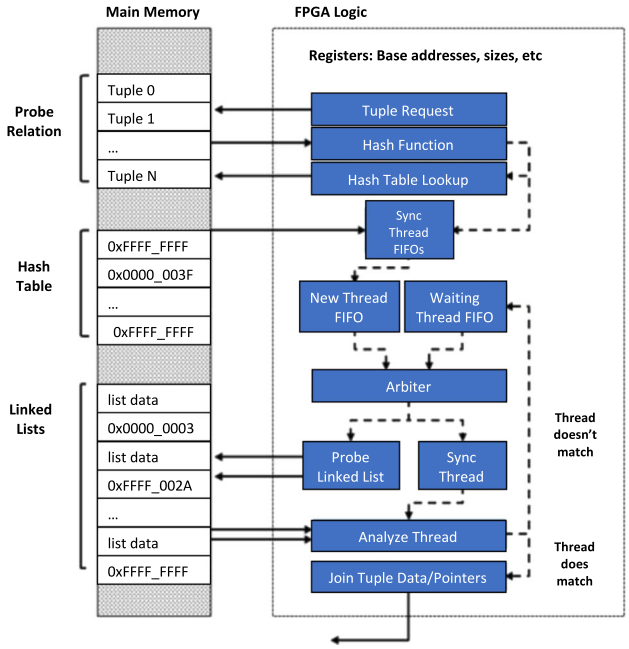


Fig. 7 The MTP probe phase engine

Figure 7 shows how the probe engine makes requests to the data structures in main memory (using 5 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 synchronizing operations. The thread only looks up the proper head pointer by probing hashed key into the table. The special 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 MTP datapath for inner join or emitted with null for left join. Otherwise, the thread is sent to the *New Thread 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 the whole chain. Therefore, threads are waiting within 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 proper node is retrieved from memory the *Analyze Thread* component determines if there was a match. Matching tuples are sent to the *Join Tuple* component. If the query is handling a right outer join, then the engine will issue a write to the hash table node to mark it as matched. This enables the final scan of right outer join to identify unmatched nodes in the build relation. If a node is the last in the bucket chain, then its thread is dropped from the datapath for inner join and emitted with null for left join. Otherwise, its next node pointer is updated in the thread's state and is sent to the *Waiting Thread FIFO*. The datapath can be easily modified to support semi-joins by including a flag in memory to indicate which tuple has been materialized already. Moreover, anti-joins can also be supported by materializing only the tuples that do not match in the build relation without any change to the datapath. 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 waiting threads, thus reducing the number of concurrent threads and ensuring that the design will not deadlock. Otherwise, when the waiting threads FIFO fills, its back pressure would stall the memory responses, causing the memory requests to stall, thus preventing the arbiter from issuing a new thread. As matches are found, the *Join Tuple* component merges the probe tuple's pointer (from the thread) with the build tuple's pointer (from the node list) and sends the result out of the engine.

3.3 Experimental results

In this section, we show how our approach is implemented on a Micron (Convey) HC-2ex architecture. We explain how the engines can be duplicated to increase parallelism and better utilize the available memory bandwidth. FPGA synthesis is known to be a time intensive process. The designs presented here are general enough to handle different join queries **without** needing to re-synthesize the FPGA logic. Our MTP implementation is compared, in terms of overall throughput, to the best multi-core approaches at the time [6]. We attempt to match the FPGA's and CPU's memory bandwidth (38.4 GB/s for the FPGA vs 51.2 GB/s for the CPU) because hash join is a memory bounded problem. Scalability and area utilization results are also presented.

MTP & software implementations The hash join approach presented in this section is implemented using the Micron (Convey) HC-2ex platform (Sect. 2.2), but the proposed methods are platform independent. The *Probe Engines* are easily ported between boards because the FIFOs (Xilinx IP Cores) are the only component specific to the FPGA and could be easily replaced with generic FIFOs. The *Build Engines* are also portable because they utilize CAMs for synchronization instead of the atomic instructions of the previous design.

Peak MTP performance depends on the number of concurrent engines, and their clock frequency. The number of engines, N, is limited by the memory bandwidth. The Micron (Convey) HC-2ex has 16 memory channels per FPGA, which run at 150MHz. During the build phase, all FPGAs are configured to hold the build engines. During the probe phase, all FPGAs are re-configured to hold the probe engines. The build engine described in Sect. 3.1 requires $M = 4$ memory channels per engine, and therefore, each FPGA can hold $N = 4$ build engines. Assuming no stalls, the peak throughput for the build phase is 600 MTuples/sec (4 engines $\times$ 150) per FPGA. Similarly, the probe engine mentioned in Sect. 3.2 uses $M = 5$ memory channels per engine, and therefore, each FPGA can hold $N = 3$ probe engines. The probe phase has a peak throughput of 450 MTuples/sec per FPGA.

The multi-core hash join approach [6] we compare against has 2 types of join algorithms: a hardware-oblivious non-partitioned joins and a hardware-conscious algorithms, which performs preliminary partitioning of their input. Both implementations perform the traditional hash join with build and probe phases; however, they differ in the way they are utilizing multi-core CPU architecture. The non-partitioned approach performs the join using the hash table which is shared among all threads, therefore relying on hyper-threading 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 cache sizes, thus 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 architecture.

Dataset description Our experimental evaluation uses four datasets. Within each dataset, we have a collection of build and probe relation pairs ranging in size from 2^{20} to 2^{30} elements. Each dataset uses the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query engines [10]. 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 a random 4-byte value. However, it could just as easily be a pointer to an actual arbitrarily long record, identified by the join key.

The first dataset, termed *Unique*, uses incrementally increasing keys which are randomly shuffled. It represents the case when the build relation has no duplicates, and thus, keys in the hash table are uniformly distributed with exactly one key per bucket (assuming simple modulo hashing). The

next dataset (*Uniform*) uses random data drawn uniformly from *uint32* value range. Keys are duplicated in less than 5% of the cases for all build relations having less than 2^{28} tuples. 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 by Gray et al. [26]. 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.

Throughput evaluation We report the multi-core results for both partitioning-based and non-partitioned algorithms. Results are obtained with a single Intel Xeon E5-2643 CPU, running on full load with 8 hardware threads. 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.) Obviously, given of the parallel nature of hash join, the CPU and MTP performance could easily be improved by adding more hardware resources.

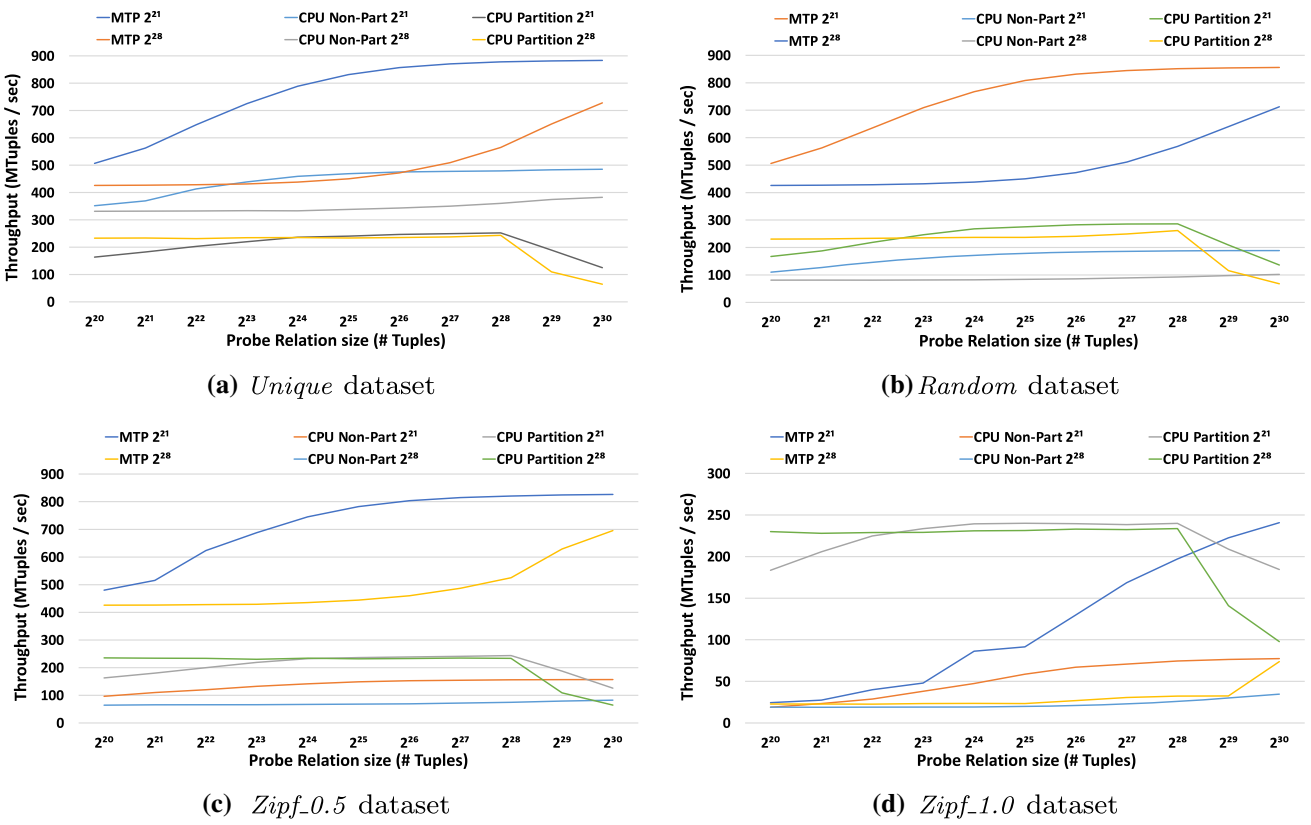
Figure 8 shows the join throughput for two build relations, with 2^{21} and 2^{28} tuples, respectively, while increasing the probe relation size from 2^{20} to 2^{30} for all datasets mentioned in Sect. 3.3. The MTP performance shows two plateaus for the *Unique*, *Uniform*, and *Zipf_0.5* data distributions in Fig. 8a–c. The MTP sustains a throughput of 820-850 MTuples/sec 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 900 MTuples/sec which can be achieved with $N = 6$ engines on 2 FPGAs. When the build phase dominates the computation, synchronization restricts the MTP throughput to about 425 MTuples/sec. (In the *MTP* 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, we can expect MTP throughput to be 500 MTuples/sec when both relations are of the same size. For the highly skewed dataset, *Zipf_1.0*, (shown in Fig. 8d) the MTP 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, thus greatly affecting the throughput.

When compared to the results in the previous work [28], the throughput numbers and trends are nearly identical. Only the extremely skewed dataset *Zipf_1.0* shows a performance hit from synchronizing in the CAM. For non-skewed datasets, synchronizing with the CAM provides platform independent performance.

The CPU results are consistent with the experiments presented in [6]. The partitioned algorithm peak performance is 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. We have seen the same pattern in the MTP case, when the throughput of the build phase is lower than the probe phase. The non-partitioned algorithm always behaves worse than the MTP 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 *Uniform* to the skewed datasets), the throughput of non-partitioned approach drops. For the highly skewed *Zipf_1.0* dataset, it falls approximately to 50 MTuples/sec. Averaging the data points within all datasets yields the following results: The MTP shows a 2× speedup over the best CPU results (non-partitioned) on *Unique* data, and a 3.4× speedup over the best CPU results (partitioned) on *Uniform* and *Zipf_0.5* data. The MTP shows a 1.2× slowdown compared to the best CPU results (partitioned) on *Zipf_1.0* data.

Figure 9 shows the throughput of the build and probe phases separately. The throughput of the build phase (blue line) is plotted for different sizes of the relation used for the built phase and is practically constant at 426 MTuples/sec. The constant throughput of the build phase is explained by the fact that the processing time of each tuple is relatively large, and hence, the system quickly reaches steady state. Therefore, the throughput is independent of the relation size.

When plotting the throughput of the probe phase, one has to consider the size of the relation used in the built phase as well. Hence in this figure, a particular probe phase plot (say "Probe Phase 2^{21}") denotes the throughput of the probe phase for various sizes of the probe relation (on the x-axis) assuming the built phase used a relation of size 2^{21}. Note that one of the two phases will dominate the throughput under different conditions and parameters (size and cardinality of the build and probe relations, and their respective distributions). The throughput of the probe phase increases with both the size of the probe relation and that of the build one. Note that the probe throughput saturates at 900 MTuples/sec which is the



(a) *Unique* dataset

(b) *Random* dataset

(c) *Zipf_0.5* dataset

(d) *Zipf_1.0* dataset

Fig. 8 Average dataset throughput of both the build and probe phase as the probe relation size is increased

maximum bandwidth supported by one memory channel per engine to materialize the results (150 MTuples/sec/engine and six engines on two FPGAs). The probe phase throughput is an example of how the available memory bandwidth for materializing the results is the limiting factor. In fact, the probe phase does not use any CAMs for caching or synchronization. In particular, processing a tuple in the build phase requires many more memory transactions than for the probe phase, and hence, the system reaches saturation soon after its initiation. The fact that the probe phase throughput, at saturation, is roughly twice that of the build phase, while the build has four engines per FPGA as opposed to three for the probe phase, indicates that processing time per tuple in the build phase is, on average, three times that in the probe phase.

Small probe relations do not have enough tuples to fill out the overall pipeline, and hence, the throughput is low because most of the execution time is spent on fill-up and drain-out. Probing on a small build relation takes less time than probing on a larger one because the size of the linked lists being traversed is smaller, and hence, the overall time is shorter and more time is spent on fill-up and drain-out. When both relations are large, there are enough tuples to keep all the engines busy for a long time so the system is at maximum throughput for a large percentage of the execution time in the

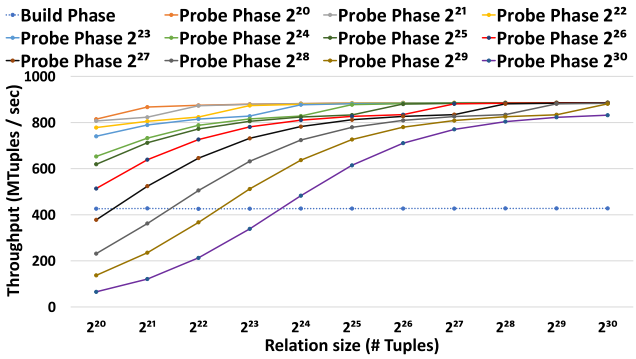
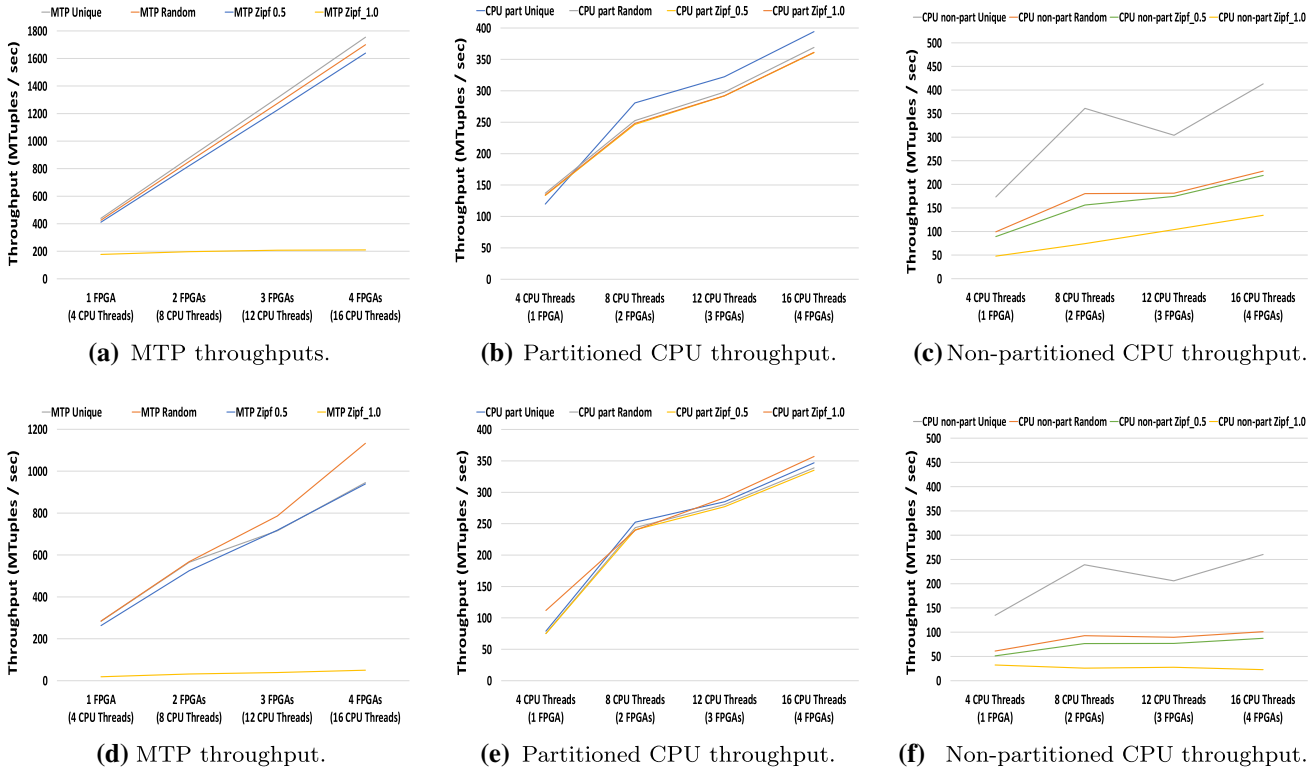


Fig. 9 Separate MTP throughput for the build and probe phases using the *Unique* dataset. The build phase throughput is nearly constant. The legend of each probe phase plot indicates the build relation size that this probe phase ran on

probe phase. Further, the time spent per tuple doing the probe is also larger because of the longer linked lists.

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 MTP implementation on FPGA. (Note that this still provides a slight advantage to the CPU in terms of memory bandwidth.) One MTP implementation refers to a single FPGA, with four engines during the build phase and



(a) MTP throughputs. **(b)** Partitioned CPU throughput. **(c)** Non-partitioned CPU throughput.

(d) MTP throughput. **(e)** Partitioned CPU throughput. **(f)** Non-partitioned CPU throughput.

Fig. 10 Throughput comparison as the bandwidth and number of threads are increased. In the top row, the build relation has 2^{21}, while probe has 2^{28} tuples. In the bottom row, both build and probe relations have 2^{28} tuples

three engines during the probe phase. We examine two cases, when the probe relation is much larger than the build one, and when they are of equal size.

Figure 10a–c shows the results when the probe phase dominates the computation. The MTP design scales linearly on datasets *Unique*, *Uniform*, and *Zipf_0.5* (Fig. 10a).

However, for the *Zipf_1.0* dataset, the performance does not scale because of the high skew. Each probe thread searches through an average of 4.8 to 6.7 nodes in the linked list. Therefore, most threads are waiting through the datapath multiple times. Having too many threads waiting limits the ability of a new threads to enter the datapath, causing back pressure and stalling. The partitioned algorithm scales as the number of threads increases but at a lower rate than the MTP approach (depicted in Fig. 10b). The non-partitioned algorithm shows a drop in performance while moving from 8 to 12 threads because of the NUMA latency emerging while moving from 1 to 2 CPUs (Fig. 10c).

The MTP scales at a lower rate when the build and probe relation are of the same size (Fig. 10d), since the throughput of the build phase scales at a lower rate and is a larger percentage of the overall runtime. However, this is significantly better scaling than our prior results, where the scaling of the build started to flatline after 2 FPGAs. The CAMs provide better scaling over more FPGAs than the atomic instructions.

Table 1 Per-FPGA resource utilization for probe and build engines

	# PEs	Registers	LUTs	BRAMs
Probe	3	106,185 (11%)	87,585 (18%)	71 (9%)
Build	4	132,288 (13%)	143,657 (30%)	40 (5%)

The slope of the scale graph is almost comparable to the CPU implementations (shown in Fig. 10e, f) again with the exception of highly skewed data.

FPGA resources utilization Table 1 shows the resource utilization (registers, LUTs, and BRAMs used) per FPGA for the MTP build phase and probe phase designs. The engines in each phase are built independently of each other. The use of CAMs in the build phase engine accounts for the higher use of LUTs in that design.

4 Using larger tuples

While the "skinny" tables with small tuples (8 or 16 byte key/value pairs) approach is valid for in-memory column-oriented databases, it is not practical for traditional DBMSs with row-major storage format. In this section, we show how our MTP design can be extended to support both wider tuples

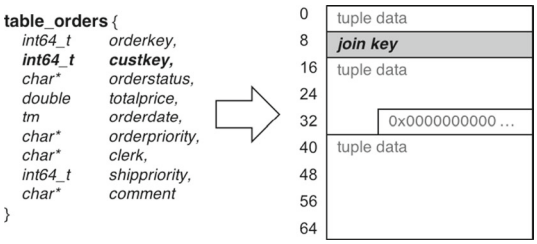


Fig. 11 Wide tuples are stored as contiguous memory blocks, but the join operation only needs the key value. Its offset can be computed at runtime

(i.e., with more attributes), and larger key length. This is done by modifying the initial memory requests to support user-specific tuple lengths, enabling the user to select a tuple length and without the need to re-synthesize the accelerator. We also increase the design's internal datapath to support wider keys and values. The updated engines are compared with modified versions of the partitioned and non-partitioned software approaches from Sect. 3. The performance of the modified join implementation is then evaluated using the TPC-H benchmark.

4.1 Supporting wider tuples

The build engine and probe engine datapaths are logically similar to those presented in Sects. 3.1 and 3.2. Only the *Tuple Request* component is modified allowing us to issue non-sequential memory requests for the join keys. This change is done to correctly handle the scenario, when tuples occupy multiple memory locations. Consider a sample tuple from the TPC-H Orders table, shown in Fig. 11. The size of this tuple is 72-bytes. Assume that we implement the join operation ($table\_orders \bowtie_{custkey} table\_customer$). In the build phase, only the join key (*custkey* which is 8-byte long) is kept inside the hash table, and in the probe phase, only the join key is needed to determine a match. Therefore, the join operation only needs to consider 8 bytes from each tuple. However, once a match is verified, the full tuple data will have to be streamed into the FPGA and merged. Since only the join key is used to identify a match, we can reuse the key/value pair model from the previous section. The value is a memory pointer to the actual tuple. To compute the stride length, runtime programmable registers are used to hold the tuple's width and the field's offset value for a given relation. In our example, the field offset is 8-bytes. This allows the MTP design to handle various tuple sizes without needing to be re-synthesized.

Using only the join key to identify matches in the probe phase can be wasteful of bandwidth when the key is not matched, for both the CPU and MTP. Furthermore, wide tuples (e.g., tuples where the key is larger than 64 bits) would require more than one memory access to fetch that tuple on a CPU. It would also require multiple operations to process that tuple since the CPU datapath is limited to 64 bits. On an FPGA, the design can be modified and adapted for a wide datapath. The fetching of the tuple would still require more than one memory access; however, its processing can be done on a custom-sized datapath.

Lastly, we need to address the issue of a join key that spans multiple words in memory. In Fig. 11, the join key is nicely placed inside a single word of memory, and thus, it requires a single memory request. Short tuple fields are usually padded with zeros to store them in memory. However, some databases (especially in-memory DBMSs) could eliminate this padding to achieve better data compression. Such optimization could cause the join key to be split across two words of memory. The modified *Tuple Request* component handles this case by issuing multiple memory requests. A reorder block is used to realign the join key before sending it out of the *Tuple Request* component.

4.2 Increasing the key/value size

As 8-byte memory words are becoming the standard, we now describe how the build engine and probe engine are expanded to support the larger key & value sizes. Increasing the key and value sizes requires the FPGA's datapath to be widened. From a design perspective, the modification boils down to simple change of variables type from $uint32\_t$ to $uint64\_t$. Throughout the HDL specification, wire sizes are increased from 32-bits to 64-bits as well. Increasing the key and value sizes also increases the memory bandwidth demands, requiring design changes since the convey bandwidth is fixed. The memory channels could be reallocated, but it will reduce the number of engines per FPGA. The FPGA engines could also duplex extra requests through the same physical channels, but it will effectively double the execution time.

With this in mind, our implementation opted to keep the number of engines per FPGA the same (4 engines per FPGA) and increased the number of memory requests over certain channels. Each tuple for the wider datapath requires one extra memory request during the build phase. The *Write Linked List* component now breaks the key/value pair into two distinct memory writes. Because this is not an atomic operation, we duplex the writes together through the same channel.

For the probe engine, earlier experiments (Sect. 3.3) showed that the architecture can achieve close to peak performance. In the updated design, a minimum of two extra memory accesses are required for the following reasons. First, in the initial design the build phase key/value pairs fit in a single word of memory, but in the updated design, they occupy two words of memory. Second, the updated design outputs two 8-byte values instead of two 4-byte values merged into a single 8-byte word. Because the probe phase requires two extra requests, two physical channels could be

duplexed: one from the *Probe Linked List* component and another from the *Join Tuple* component. However, duplexing any memory channel doubles the number of memory requests and will cut the throughput performance in half. A better option is to reallocate the memory channels per each FPGA, so that the updated design uses 7 channels per engine. As a result, only two updated probe engines will fit in the 16 available memory channels for the Micron (Convey) FPGAs, whereas the previous design (Sect. 3.3) could fit three probe engines with the same channel budget. Therefore, the throughput performance is decreased only by one-third.

4.3 Adjustments to software approaches

Although the memory is byte-addressable for both hardware and software join implementations, CPUs incur a special kind of memory access reading in the whole L1 cache line (64 bytes on our architecture) and bringing it into the cache to take advantage of the locality. However, increasing the tuple length cannot utilize any of the locality properties, neither temporal (tuples are read only once both during build and probe phases) nor spatial. (Size of the tuple typically is greater than a single cache line.) Thus, the extensive caching, which could not be disabled, only puts additional pressure on memory bandwidth, effectively deceasing the processing throughput.

Because such penalty should be paid each time the join key is accessed, we choose to implement a projection step and materialize the intermediate result in a form of key/value pairs, with the same format used in Sect. 3.3. Projection step essentially allows us to amortize join key access overhead by allocating additional memory to store projected relations. Our initial experiments showed that executing projection as a separate step, without incorporating it into build or probe phase, yields better performance results due to better locality.

4.4 Experimental results

All results obtained in this subsection were collected by running tests on the Micron (Convey) HC-2ex platform. In Sect. 3.3, we showed that MTP design performance is predictable for most relations, the exception being highly skewed datasets. However, these experiments used "skinny" key/value relations. In this subsection, we show that the same MTP design predictability holds for larger real world datasets. We extract the join operation from a query in TPC-H that we use as a proxy. Results show the join performance on two tables (Orders and Customer) from the TPC-H benchmark suite with varying scale factors.

TPC-H dataset For our wide tuple experiments, we used the TPC-H benchmark from the Transaction Processing Performance Council. It is a decision support benchmark which is traditionally used to compare analytical query performance

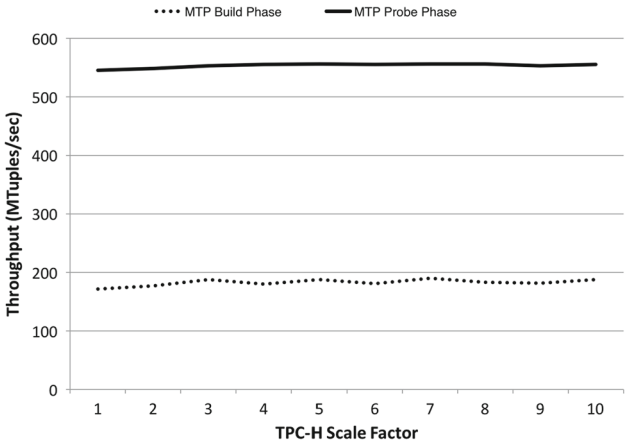


Fig. 12 MTP design throughput results for the build and probe phases as the TPC-H scale factor is increased. Results are obtained using two tables (Orders and Customer) from the TPC-H benchmark suite

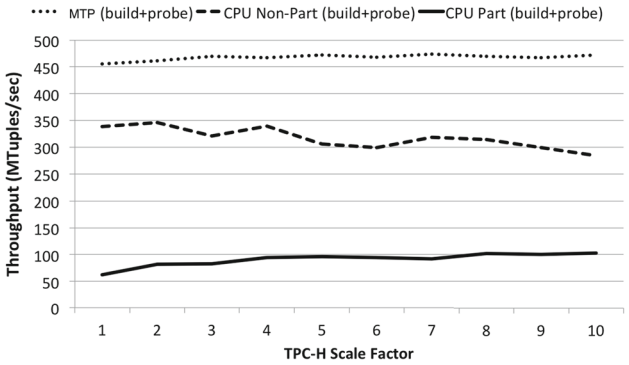


Fig. 13 MTP and CPU total throughput performance as the TPC-H scale factor is increased. Results include both the build and probe phase execution times. Results are obtained using two tables (Orders and Customer) from the TPC-H benchmark suite

on commercial database systems. The schema consists of 8 different relations, and 22 unique queries. A scale factor (SF) is used to control the datasets' size and allows it to generate relations between 1GB and 100TBs. While the absolute size of the relations might vary, their relative sizes are fixed. For example, the size of the Orders table is 10 times bigger than the size of Customer relation, regardless of the SF.

The MTP design can offer significant speed up for selection and projection. Our experiments perform the $Orders \bowtie_{custkey} Customer$ operation, which is used in queries $Q3$, $Q5$, $Q7$, $Q8$, $Q10$, $Q13$, and $Q18$. For all tests, the smaller *Customer* table is used as the build relation, and the *Orders* table is used as the probe relation. We use the *qGen* utility to generate 10 datasets with the scale factor varying between 1 and 10.

Throughput results As in Sect. 3.3, we compare the throughput results between one CPU and two FPGAs, in effort to match the memory bandwidth.

Figure 12 shows the MTP design throughput results for the build phase and probe phase separately. We see that the build phase is still bottlenecked by synchronization as it was in Sect. 3.3. Peak performance for 8 engines at 150MHz and one duplexed memory channel per FPGA is 600 MTuples/sec, while sustained performance is around 200 MTuples/sec.

The probe phase achieves near peak performance (about 550 MTuples/sec), again in line with the experiments from Sect. 3.3. Because of the increased key/value pair size, only 4 engines could fit on two FPGAs, and thus, the combined peak theoretical throughput is 600 MTuples/s.

In Fig. 13, we show the end-to-end throughput for all three approaches: MTP design, CPU partitioned, and CPU non-partitioned. In the TPC-H benchmark, the probe table is 10× larger than the build table and therefore dominates the computation time. This does not have a big impact on both CPU approaches, but for the MTP design, it skews the performance toward the probe phase's throughput. The MTP design achieves throughput results between 450 MTuples/sec and 475 MTuples/sec depending on the scale factor.

The non-partitioned CPU algorithm achieves better throughput (between 300 and 350 MTuples/sec) in comparison with partitioning-based one (between 50 and 100 MTuples/sec). In the Customer relation, each join key is encountered exactly once, and therefore, its key distribution is identical to the *Unique* dataset from Sect. 3.3. As explained earlier, this is why the non-partitioned approach outperforms its partitioning-based counterpart.

Conclusions We have demonstrated how the MTP hash join designs can be extended to support a more generic tuple formats. We proved that our design is not only limited to column-major storage formats, but can offer performance improvements in traditional row-based DBMSs. While both the CPU and MTP design dropped in performance, our results showed that the MTP design still holds an edge over both the partitioned and non-partitioned CPU approaches.

5 MTP group-by aggregation

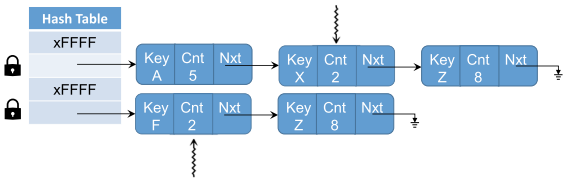
In our group-by design, we assume the input relation fits in main memory but is too large to fit locally on the FPGA's memory. The aggregation engine is implemented with a single memory channel per engine (PE) $M=1$ and $N=12$ PEs. As the results will show, we found that condensing all engine requests onto a single channel and multiplexing them internally yields the highest utilization. This prevents stalls in one system component from stopping progress on multiple memory channels. It also increases the inter-engine parallelism. The choice of parameters M, N is further discussed in Sect. 5.3.1.

The mixed read-write nature of aggregation in conjunction with multiple outstanding memory requests requires explicit synchronization to ensure correctness. Atomic operations are one option, but this approach severely impacts the performance. Moreover, unlike the join operator, aggregated tuples may exhibit temporal locality.

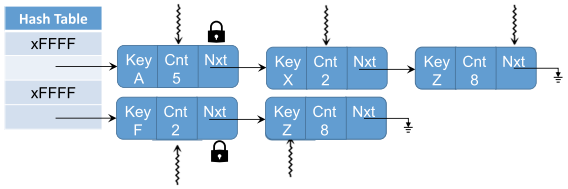
We propose a novel multithreaded aggregation implementation based on CAMs [1] extended with fine-grained locks and efficient memory channel allocation. The design leverages explicit synchronization combined with the caching properties of the CAM. This fits perfectly in the context of group-by aggregation: Firstly, the latency of a single aggregation thread is hundreds of cycles, which means many threads can have identical keys. With a CAM, we can merge these threads pre-aggregating the result locally on the FPGA and reduce the number of outstanding memory requests. This merging is achieved by leveraging caching properties of the CAM (allowing us to hold the aggregate value for a particular key): We call this the Filter CAM. It also allows us to alleviate skewed data distributions, where a subset of values appears as duplicate more often than the rest. Secondly, CAMs allow the FPGA to enforce locking on specific memory addresses and therefore decrease the granularity of the locks and boost the performance: We call this the Lock CAM. Each group-by aggregation engine uses one Filter and one Lock CAM. In the following description, we assume a COUNT aggregation as an example. In the original aggregation design [1], our locks were implemented at the granularity of hash table buckets (Fig. 14a). This guaranteed that only one thread was working on a list in the hash table at a time and it was free to modify the list as needed. With exclusive access to the list, the threads can perform node inserts in sorted order to improve the merge phase. However, such coarse-grained locking has a big impact on the parallelism the system is able to achieve. This is especially noticeable on skewed datasets where a majority of keys might map to the same bucket. All of those threads must stall and wait for the previous thread to finish. And the wait for each thread increases hundreds of cycles for each node added to the list. Each thread must pay this penalty even if it is only going to increment the count in a node and not modify the list structure.

5.1 Fine-grained locks

The first insight motivating the fine-grained locks (FGL) design comes from the benefits of the top-level Filter CAM. All new tuples that enter the aggregation start at the Filter CAM. If the key already exists, its count is incremented in the Filter CAM and the thread terminates. If the key does not exist in the Filter CAM and there is space, the key is added and a thread starts the hash table search. This construction guarantees that all threads searching the hash table are unique—they will never try to update the count in the same node because they all have different keys. We can take advantage of this design and move the lock lower to node pointers (Fig. 14b).



(a) Coarse-grained locks stop threads at the bucket level and prevent concurrent searching through the linked list.



(b) Fine-grained threads lock at the node level, increasing thread concurrency and only synchronize structural changes.

Fig. 14 Coarse-grained versus fine-grained locking

This enables synchronizing where it matters, when a thread wants to insert in the list and make a structural change.

Lock free reads Since we only lock for structural changes (i.e., inserting a node), that means only writes need to be protected. This observation raises the question—*is it safe for threads to read past a lock?* Consider the possible situations of a thread progressing down a list (Fig. 15). $\boldsymbol{T_1}$ is a thread searching the linked list for a key $\boldsymbol{\beta}$. $\boldsymbol{T_0}$ is a thread inserting new node with key = $\boldsymbol{C}$ and count = 2. Therefore, the next node field at node 1 is locked for the insertion of a pointer to the new node containing key = $\boldsymbol{C}$ and count = 2. Since the link-list has key-values sorted, there are three possible outcomes of $\boldsymbol{T_1}$ traversing the linked list:

1. $\boldsymbol{\beta > X}$ There is no conflict between threads $\boldsymbol{T_0}$ and $\boldsymbol{T_1}$. Hence, the lock on the next node at Node 1 is irrelevant and it is safe for $\boldsymbol{T_1}$ to proceed. $\boldsymbol{T_1}$ will either find a node with a key equal to key $\boldsymbol{\beta}$, or it will need to insert a new node with $\boldsymbol{\beta}$ as the new key later in the list.
2. $\boldsymbol{\beta = X}$ In this case, $\boldsymbol{T_1}$ has found its node, which is Node 2, and can update the count without synchronization.
3. $\boldsymbol{\beta < X}$ Thread $\boldsymbol{T_1}$ has progressed too far and must insert. However, insertion is gated by the lock on Node 1. $\boldsymbol{T_1}$ will safely synchronize on that lock and will try again after the lock is free.

There are several benefits to using this design where reads are not locked. First, at no point does a thread need to stall until it needs to insert. If reads had to wait for the lock, these fine-grained locks could deteriorate to behavior like the coarse-grained lock—a thread blocks the start of the list and

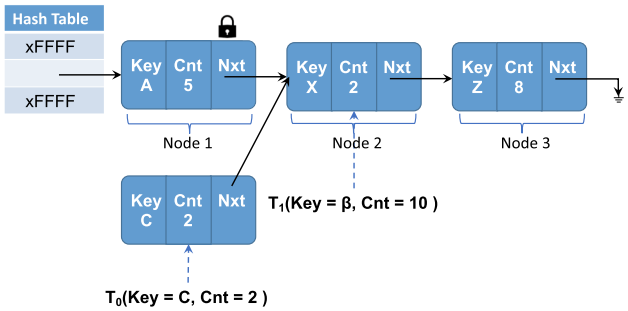


Fig. 15 Demonstrating lock free reads. While new node is being inserted between Node 1 and Node 2, thread $\boldsymbol{T_1}$ can traverse the liked list looking for $\boldsymbol{\beta}$. If that key is found in the linked list, the count is updated; otherwise, a new node is inserted

all work stops. However, the more important benefit comes from the behavior of the aggregation algorithm. For any given aggregation, the cardinality of the key can be much smaller than the size of the data table that is being scanned. Therefore, there will be a contentious period at the beginning of the execution where keys are getting inserted. However, once all keys have been seen once and inserted in their respective locations in the table, the rest of the execution will proceed lock free. In the case, where the cardinality is on the order of the number of items in the data table, there may be no lock-free execution. However, this is still an improvement over coarse-grained locks since there will still be parallel inserts on the linked lists. As an example, consider the worst case dataset where all keys hash to the same bucket. With coarse-grained locks, every item will be serialized for insertion by locking the head of the list to do the update. With fine-grained locks, the number of insert locations increases over time, decreasing contention and increasing the parallelism. If the keys are sorted, such that each ordered thread needs to append to the list, we lose the parallelism on insert, but we still gain in the parallel searching of the list for threads to find their insert location.

5.2 Aggregation engine workflow

Our design of an aggregation operation uses a custom hardware datapath called *aggregation engine*. Initially, each tuple from the relation is streamed from memory, gets assigned to a separate MTP thread, and starts its pipelined execution. Figure 16 shows the state diagram for a single thread inside the aggregation engine. The Filter CAM is used to merge threads with identical keys, hence reduces the memory request contention, and minimizes the synchronization overhead. When there is a match in the Filter CAM, the thread will increment the key count in the Filter CAM. The thread that originally created this entry in the Filter CAM will update the HT with the new key count. However, due to hash collisions, the synchronization cannot be avoided completely; thus, the Lock

Table 2 Contents of the Filter CAM, Lock CAM and hash table (HT) and *modifications* altering all of them, while the following keys are processed: A, C, A, B, A

Cycle	Key	Filter CAM	Lock CAM	Hash Table	Comments
1	A	*Miss, Insert (A,1)*	{}	{}	Request to search key A in HT is sent
		{(A,1)}			
2	C	*Miss, Insert (C,1)*	{}	{}	Request to search key C in HT is sent
		{(A,1),(C,1)}			
3	A	*Hit, Update A*	*Miss, Thread 1 locks hash(A)*	{}	Key A not found in HT, $Bucket_{hash(A)}$ is locked
4		*Thread 1 clears key A*	*Hit, hash(A)=hash(C)*	{(A,2)}	Create new entry (A,2) in HT
		{(C,1)}	{hash(A)}		Key C not found in HT, *Thread 2* waits for lock
5	B	*Miss, Insert (B,1)*	*Thread 1 frees lock on hash(A)*	{(A,2)}	Request to hash(C) in HT is sent
		{(C,1),(B,1)}	{}		*Thread 2 restarts at previous address*
6	A	*Miss, Insert (A,1)*		{(A,2)}	Request to search key B in HT is sent
		{(C,1),(B,1),(A,1)}	{}		*Thread 2 reaches end of list*
7			*Thread 2 locks* $node(A).next$	{(A,2)}	Request to search key A in HT is sent
		{(C,1),(B,1),(A, 1)}	$\{node(A).next\}$		$node(A).next$ is locked
8		*Thread 2 clears key C*	*Thread 2 frees lock for key C*	{(A,2),(C,1)}	Create new entry (C,1) in HT
		{(B,1),(A,1)}	{}		Key B not found in HT, CAM busy
9		*Thread 5 clears key A*	*Thread 4 locks hash(B)*	{(A,3),(C,1)}	Key A found
		{(B,1)}	{hash(B)}		Update key A in HT to (A,3)
10		*Thread 4 clears key B*	*Thread 4 frees lock for key B*	{(A,3),(C,1),(B,1)}	Create new entry (B,1) in HT
		{}	{}		

Assume hash(A)=hash(C). Initially, both CAMs are empty. Filter CAM maintains the occurrence of duplicate keys, while Lock CAM locks linked-list next pointers

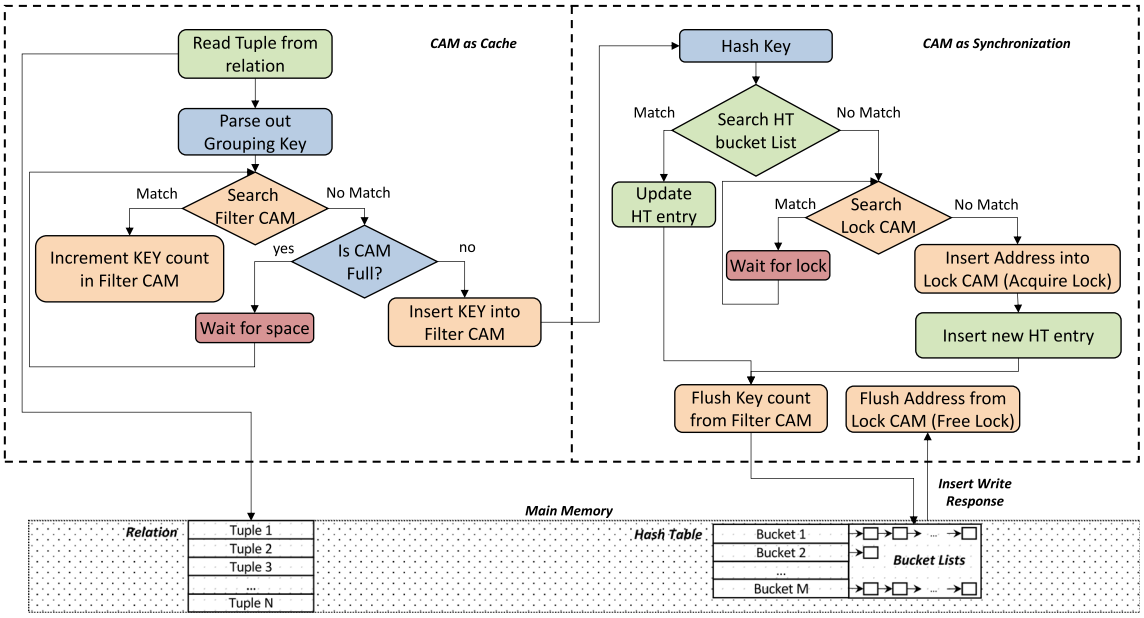


Fig. 16 A state diagram for threads in the aggregation engine

CAM is used to acquire locks on the hash table, ensuring its integrity. Each engine uses its own CAM for synchronization. As a result, values are aggregated in separate hash tables, which requires an extra merging phase at the end of the computation performed by the FPGA. Merging overhead grows as we increase the number of engines per FPGA, but it is an overhead that is amortized as the size of the dataset grows.

Table 2 shows an example of events and contents of Filter CAM, Lock CAM, and main memory hash table, while the input stream consists of 5 tuples with the following keys: A, C, A, B, A. The design assumes the COUNT aggregation function, and thus, the Filter CAM maintains an occurrence count of duplicate keys. However, other functions could be potentially applied. Note that operations updating the CAMs are performed immediately, whereas main memory hash table accesses (e.g., search, entry update, entry insert) take hundreds of cycles to finish. For example, *Thread 1* sends a request to search value A in a hash table and gets response only at $Cycle_3$. Lock CAM maintains the locks for all addresses which are currently being modified. Notice that all threads only acquire locks after searching memory. Locks are only needed when creating a new node. Even though both *Thread 1* and *Thread 2* need to search the same bucket, they will not synchronize until after finding the list is empty and trying to add a new node. *Thread 1* finishes first and is able to get the lock, and *Thread 2* finds it must wait in the next cycle. Once a thread completes, it invalidates the record in both CAMs and frees up resources for other threads. Threads, waiting for a place in a CAM, will continually cycle through a FIFO until the resource is available. Whenever there is a hit in the Lock CAM, the thread waits until the lock is released, e.g., *Thread 2* resumes its work only at $Cycle_5$. *Thread 3* provides an example of early termination, because its value was locally aggregated in Filter CAM in $Cycle_3$. After $Cycle_5$, we see more concurrency as 3 threads are searching the list. *Thread 2* provides an example of fine-grained locking in $Cycle_7$. The thread gets to the end of the list and locks the next pointer of node A. At the same time, *Thread 5* is able to find node A in the list and update its count without any locks. Finally, *Thread 4* is able to finish in $Cycle_{10}$ after a long memory request and waiting for a free cycle in the Lock CAM. This code can be adapted to fit other forms of aggregation operations such as SUM(), AVG(), MAX(), and MIN(). In order to support MAX function instead of COUNT, the function in the Filter CAM needs to change from performing increments by 1 to computing MAX (current_max, new_value). Both functions would require reading the current value from memory (COUNT or MAX). While the COUNT always updates the count in memory, MAX may or may not update the value in memory. As for AVG(), the Filter CAM is still needed to count and then divide by number of elements. On the software side, an opcode can be used to activate a different function on the FPGA.

5.3 MTP design optimizations & trade-offs

The main factor limiting performance of this memory-bounded problem is the efficient use of available memory bandwidth. In this paper, we use a Micron (Convey) HC-2ex machine, but our designs are platform independent. In the HC-2ex, the communication between the FPGA and main

memory relies on the abstraction called *channel*. Each channel supports independent and concurrent read/write accesses to memory.

The original design of our aggregation engine required 4 memory channels: one for streaming the input tuples, one for accessing the in-memory hash table, and finally two channels for the bucket lists read/write operations. Since the Micron (Convey) HC-2ex has 16 memory channels, we replicated 4 engines ($\frac{16}{4}$) on a single FPGA thus leveraging inter-engine parallelism. Our original experiments showed that some memory channels were idle for almost 70% of the total execution time. Since the channels within an engine are statically assigned to perform different functions of the pipeline, back pressure from some components (e.g., thread waiting through CAM synchronization) introduces stalls and decreases the effective throughput.

In order to increase memory utilization, we then *multiplexed* a pair of engines on the same set of memory channels, thus allowing the same channel to be used by two different engines. This means that the following engine operations: (1) send and receive tuple request and response, (2) read and write respective values to the hash table, (3) read and write entries into respective bucket list, can run concurrently on two different engines. The multiplexed design increases the number of CAMs that could be placed on the FPGA, leading to further improvement in throughput. Unlike the original design [1], the new multiplexed engine uses 5 memory channels (adding an extra channel for accessing the in-memory hash table). This enabled 6 engines ($2 * \lfloor \frac{16}{5} \rfloor$) on a single FPGA.

In this latest design, to further improve the utilization of memory bandwidth, we have reduced the number of memory channels down to 1 per engine. For each engine, all requests for streaming in the tuples, accessing the hash table, and accessing linked lists are multiplexed internally. This construction enables up to 16 engines per FPGA; however, due to routing constraints, we only implemented 12 engines. While multiplexing more requests over a given channel likely increases the latency of any given thread, it enables the engine to always have a request available to issue and can keep the engine in the steady state longer. If the aggregation datapath is working through requests but cannot take any more tuples from the stream, it is a waste of bandwidth to dedicate a channel to streaming tuples. As the percentage of execution time in the steady state increases relative to the fill-up and drain-out stages, the overall throughput of the system increases.

5.3.1 Experimental results

The MTP aggregation implementation is compared in terms of overall throughput against the best multi-core approaches [16,59] running on a single processor with 4 parallel threads. A summary of the various software aggregation algorithms follows, as well as a description of the datasets used in the experiments.

Software implementations In order to evaluate our MTP architecture, we have implemented the following state-of-the-art multithreaded software aggregation algorithms: (i) Independent Tables [16], (ii) Shared Table [16], (iii) Hybrid Aggregation [16], (iv) Partition with Local Aggregation Table [59], and (v) Partition and Aggregate [59]. Here, (i) and (ii) are considered as non-partitioned approaches, while (iii) and (iv) are hybrid, and (v) is a partitioned approach.

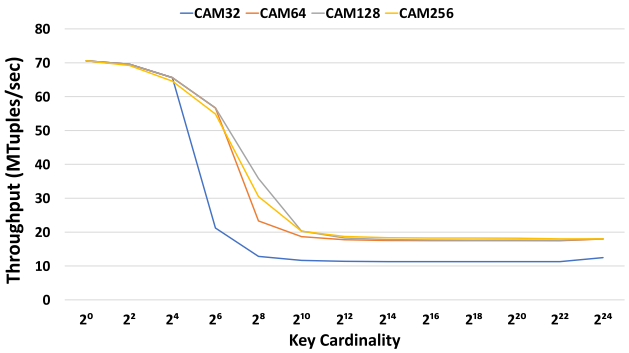
- **Independent Tables** [16] is the approach most similar to our hardware implementation. The tuples are evenly split among separate software threads (without partitioning), and each thread aggregates result into its own hash table. Once the aggregation is complete, all tables are merged together, which requires write synchronization.
- **Shared Table (with locking or atomic synchronization)** [16] splits the tuples evenly between threads, but all threads aggregate their results into a single hash table, and hence, no extra merge step is required. The algorithm could use different synchronization primitives: either pthread mutex implementation or Intel-specific hardware atomic instructions. Preliminary experiments showed that atomic primitives are significantly better on low key cardinalities and do not have any difference from mutexes on medium and large cardinalities, so we choose atomics as a default synchronization primitive in all further experiments.
- **Hybrid Aggregation** [16] is a combination of two previous approaches. This algorithm allocates a small hash table for each thread. The size of the table is calculated based on the processor's L2 size to avoid cache misses. If the local table has enough space for a new value, or the value already exists in the table, that tuple is locally aggregated. Once the local table is filled and the next tuple requires a new slot, the oldest entry in the cached table will be spilled into larger shared table, residing in main memory, thus maintaining only "hot" data in L2 cache. Once aggregation is complete, all small cached tables are merged into the large shared table. Merge step is synchronized as in *Independent Tables* case.
- **Partition and Aggregate** [59] (also known as count-then-move [17]) uses individual tables per thread, but before aggregation is performed the tuples are partitioned, in contrast to all aforementioned approaches. Separate partitioning step makes sure that all threads will work on non-overlapping values, and hence, aggregation could be done without any synchronization, and the final tables are simply concatenated, rather than merged. As with the partitioned join implementations, *radix clustering* algorithm is a backbone of this preliminary step.

– **PLAT (Partitioning with Local Aggregation Table)** [59] is a combination of two previous techniques. The algorithm takes advantage of the fact that we are performing an additional data scan, while doing a pre-processing step. While partitioning tuples into groups with mutually exclusive keys, each thread tries to aggregate values into its own small L2-resident table, as in *Hybrid Aggregation* approach. Values that do not fit into the small table are partitioned using *radix clustering* algorithm. Once preprocessing is done, standard lock-free aggregation is applied. To finish, all tables which were produced during aggregation are concatenated together, while local tables are synchronously merged.

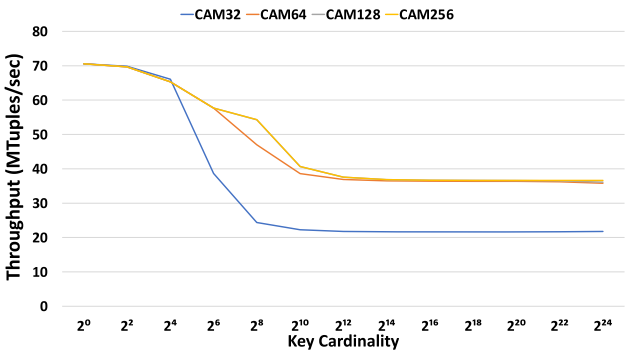
Dataset description We use five datasets with various key distributions, namely: Uniform, Heavy Hitter, Moving Cluster [16], Self-Similar, and Zipf_0.5.

- In the **Uniform** dataset, all key values are picked from the *uint64* key range with uniform probability. After that generated key/value pairs are randomly shuffled.
- A half of the tuples in the **Heavy Hitter** dataset [16] share the same a key value. The remaining key values are picked uniformly and evenly distributed throughout the the entire relation.
- In the **Moving cluster** dataset [16], tuples are grouped into clusters depending on their key values. Lower key values are more likely to appear at the beginning of the relation, whereas tuples with higher key values tend to appear at the end of the relation.
- **Self similar** uses Pareto rule to model key distribution in a dataset: A single key value is shared by 20% of the tuples. Of the remaining 80% of tuples, 20% of those share another key value. This process is repeated recursively to generate the relation. Tuples are randomly shuffled. The generation algorithm is described by Gray et al. [26].
- In the **Zipf** dataset, key values follow the Zipf distribution with a skew coefficient of 0.5. The generation algorithm appears in aforementioned work [26].

Each dataset consists of several benchmarks with cardinalities ranging from 2^{10} to 2^{22} unique keys. The relation size in all of the experiments was 256 million tuples (in line with previous research [59]). Each dataset used the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query processing algorithms [5,9,10] and represents a popular column-wise storage format. The first 4 bytes of the tuple hold the unique primary key, while the rest is reserved for the grouping key. Since we are only interested in counting records with the same grouping keys, our tuples do not store any other information. However, none of the design choices prevent the use of "wide" tuples (i.e., containing fields other than primary and grouping keys).



(a) Uniform



(b) Heavy Hitter

Fig. 17 Aggregation throughput of single engine for 256M tuples as the Filter CAM size is varied from 32 to 256

This could be easily supported by adding a key extraction component into the MTP design. Moreover, experimenting with such "skinny" tuple format yields the best performance for software implementations, since it minimizes the cache capacity misses, which would decrease caching effectiveness otherwise.

Effect of filter CAM size in MTP group-by aggregation The throughput of a multithreaded engine is determined by the number of threads needed to fully mask latency. In this FGL engine, one of the key controls on the number of threads concurrently working is the size of the Filter CAM. Since every entry in the Filter CAM starts a thread searching the hash table for a node, we started by experimenting on the effect of the CAM size on throughput. Figure 17 shows the throughput as a function of the Filter CAM size for two of the data distributions and illustrates the caching aspect of a CAM. The other distributions show similar behavior.

For the uniform distribution (Fig. 17a), there is a sharp drop in throughput where cardinality grows larger than the Filter CAM size. As the cardinality increases, there will be little temporal locality in the tuple stream and pre-aggregation provides little help. CAM sizes of 64 or 128 provide similar throughput, especially for larger cardinality where hash table searching dominates.

For the Heavy Hitter distribution (Fig. 17b), there is a similar drop in throughput where cardinality grows larger than the Filter CAM size. However, in this instance the Filter CAM size definitely affects the achievable throughput when hash searching dominates. A CAM size of 32 keeps the throughput similar to the uniform distribution. CAM sizes of 64 and above nearly double the throughput as the Filter CAM is able to exploit locality in the skewed data. As with all caches, there are diminishing returns for increasing the size. Maintaining locality for larger cardinalities requires significantly larger CAMs. Considering the diminishing returns of CAM sizes above 64, our current design uses a Filter CAM size of 128 to ease resource pressure.

Throughput evaluation

Figure 18 shows the throughput of group-by aggregation as the key cardinality is increased, obtained for various distributions. Throughput was measured for two MTP engine designs: multiplexed [1] and FGL, and five software implementations: two non-partitioned, two hybrid, and one partitioned. Throughput for the skewed Heavy Hitter dataset in Fig. 18d resembles the results for Self-Similar dataset in Fig. 18b, while the throughput for moderately skewed data Zipf_0.5 18e is similar to the results obtained for Uniform dataset in Fig. 18a. Software implementations demonstrate the best performance on Moving cluster dataset in Fig. 18c due to the property of the data distribution: Similar grouping keys appear in the input stream clustered together, increasing CPU-cache hit rates.

Despite all the differences in data distribution, the CPU aggregation performance is mainly determined by the dataset's key cardinality. When the number of unique keys is low, hash tables can fit into the CPU cache entirely. However, as the cardinality increases, cache misses start to hamper the throughput due to high latency memory round-trips. Software performance severely deteriorates at cardinalities higher than 2^{18} on all datasets for all algorithms. Another trend, which appears in all experiments, is that the Independent Tables approach yields the best result across all software algorithms. Nevertheless, that algorithm exhibits poor scalability, since the amount of memory needed for aggregation processing grows linearly with the number of parallel threads and the key cardinality. As the number of parallel threads increases, the amount of available memory could quickly become a bottleneck. We could also see that hybrid algorithms (PLAT and Hybrid Aggregation) outperform traditional partitioned (Partition and Aggregate) and non-partitioned (Shared Table) approaches by amortizing the cache miss cost and sustain a throughput around 400 MTuples/sec. This trend continues for cardinalities up to 2^{16}, which marks the end of L3-cache residency. After that point, the performance advantage of hybrid algorithms vanishes and drops below 100 MTuples/sec.

The throughput of the MTP designs also drops as the key cardinality increases; however, this effect is much less pro-

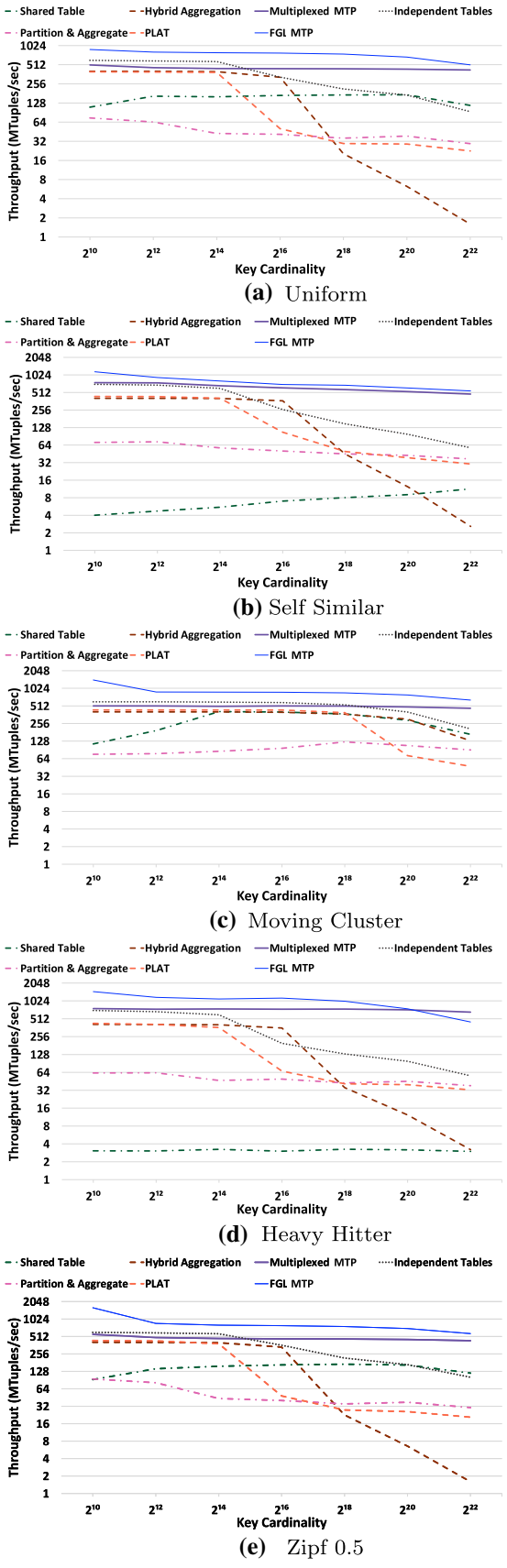


Fig. 18 Aggregation throughput of hardware and software approaches for datasets with 256M tuples. *Y*-axes are logarithmic

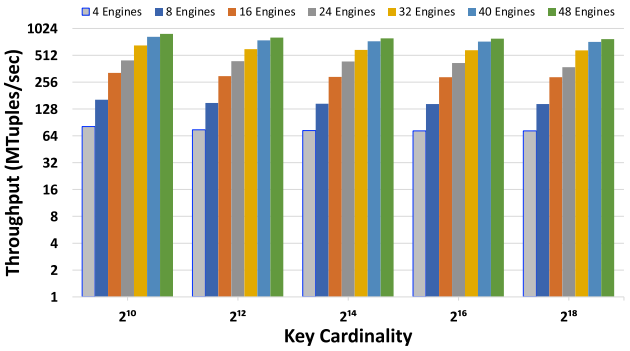


Fig. 19 The effect of varying the number of engines on throughput from 4 to 48 engines on all 4 FPGAs for Uniform key distribution dataset with 256M tuples. *Y*-axis is logarithmic

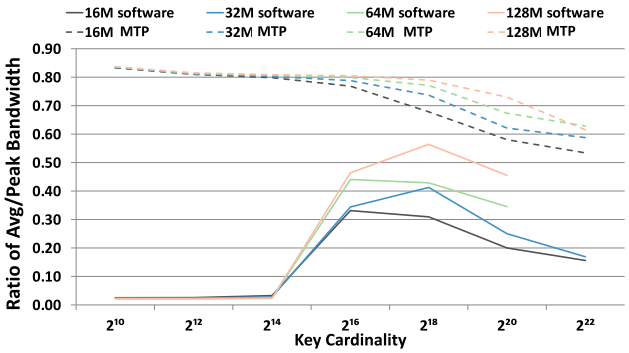


Fig. 20 Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the FGL design for varying dataset sizes and key cardinalities

found. Unlike the software throughput, this drop is explained by the overhead, introduced by the post-processing merge step.

The most striking aspect of the MTP throughput is that it is mostly constant across the range of cardinality dropping by a factor of two at the most, while the software one drops by factors in the 100s. The results clearly show the benefits of the FGL MTP design over the multiplexed one. An important detail to note is that the FGL implementation is only using 12 engines, thus only using 75% of the available memory bandwidth. Even with this handicap in bandwidth, FGL design is able to outperform the multiplexed design. The increased throughput comes from three main factors. First, reducing the number of channel allocations lets us use more engines and hence see increased inter-engine parallelism. Second, because we are multiplexing more requests over the same channel, this design is able to use the available bandwidth more efficiently. This better efficiency improves the latency-masking and increases throughput. Last, the fine-grained locks mean that we only need to do locking in the beginning of execution, while the first nodes are being inserted. The remainder of the execution is lock free.

Trade-offs For a given workload, as the number of engines in the machine increases, the work done in each engine decreases and the time spent by this engine in the steady state, at maximal throughput, is reduced, and hence, the overall throughput is reduced as more time is spent in the fill-up and drain-out stages.

We varied the number of engines per FPGA from 1 to 12 (4-48 total engines) and tested it with Uniform key distribution on 256M tuples dataset. The results in Fig. 19 show that the throughput is linear with the number of engines. As the number of engines grows higher, the bandwidth starts to saturate, causing the gain in throughput to decrease. The gain in throughput when going from 40 engines to 48 engines is less than that when going from 32 engines to 40 engines.

The comparison of the FGL throughput to that of the multiplexed design appears in Fig. 18. The following factors contribute to the decline in throughput:

1. The locking ratio increases with cardinality, taking up a much larger portion of the execution time.
2. FGL MTP design has 12 engines versus 6 in the multiplexed MTP design, which means each engine has less data to work on.
3. FGL MTP design has a smaller Lock CAM of depth 32 compared to 128 in the multiplexed MTP design. It was not possible to place and route 12 engines with the larger CAMs on the Xilinx Virtex6-760 FPGA.

We note that a newer FPGA could fit a larger CAM and 12 engines eliminating the drop in throughput.

Discussion The performance benefits of the MTP approach come not from architecture-specific features of the FPGA devices, but from the massive multithreading that enables the MTP engine to better utilize the available memory bandwidth while masking latency. Figure 20 depicts the ratio of effective average memory bandwidth to peak theoretical memory bandwidth for the best software (Independent Tables) and the MTP implementations while varying the dataset sizes and key cardinalities. The MTP approach allows the FGL MTP to keep the ratio almost constant, irrespective of the dataset size or key cardinality. Since the FPGA's and CPU's memory bandwidth are 38.4 GB/s and 51.2 GB/s, respectively (Sect. 3.3), then ratio 1 in Fig. 20 corresponds to 38.4 GB/s on the HC-2ex and 51.2 GB/s on the CPU.

For the software implementation, at low cardinality the aggregated relation and hash table are mostly in the cache hierarchy and there are almost no memory accesses, and hence, the ratio approaches zero. The ratio peaks at around 0.5 for cardinality 2^{18}, but drops significantly for higher key cardinalities.

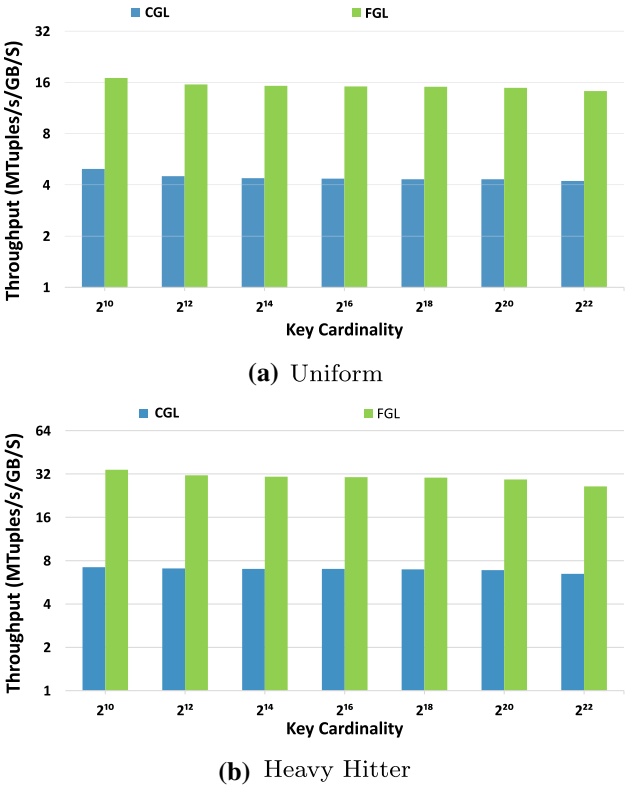


Fig. 21 Aggregation throughput normalized against available bandwidth of CGL and FGL approaches for Uniform key and Heavy Hitter key distribution datasets with 256M tuples

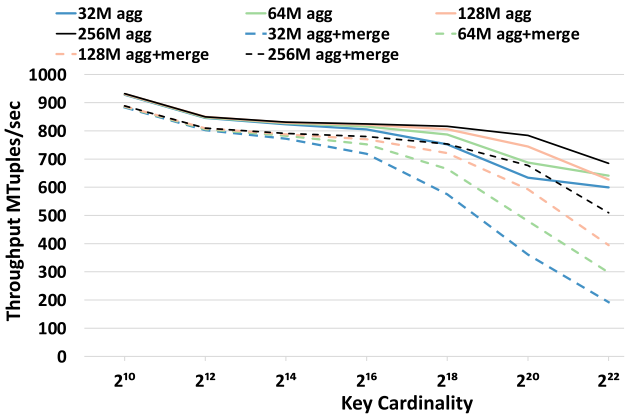


Fig. 22 Effect of varying relation sizes on the MTP aggregation throughput for datasets with Uniform key distribution. Solid lines represent throughput of the aggregation step (without merge operation), while dashed lines represent end-to-end (aggregation followed by the merge) throughput

Table 3 Per-FPGA resource utilization for aggregation engines

	# PEs	Registers	LUTs	BRAMs
Original	1	99,597 (11%)	87,194 (18%)	126 (17%)
MUX	6	179,641 (18%)	200,175 (42%)	250 (34%)
FGL	12	240,118 (25%)	296,778 (62%)	192 (26%)

The effective memory bandwidth of the CPU implementation tends to grow as the size of the relation increases (from 16M to 128M), whereas the MTP approach is less susceptible to data size variations. For average cardinalities, the FGL MTP implementation is almost 30 times higher.

For very large cardinalities, the FGL MTP implementation ratio drops due to the small Lock CAM as explained in Sect. 5.3.1. Yet, the ratio is still about 2.5 times higher on average than the software.

Fine-grained locks versus coarse-grained locks The performance benefits of FGL compared to the CGL [1] are shown in Fig. 21a (for the Uniform key distribution dataset) and 21b (for the Heavy Hitter key distribution dataset). For this experiment, both FGL and CGL designs have the same number of engines and the same sizes of Filter and Lock CAMs. The throughput is normalized over the available bandwidth. The FGL design normalized throughput is 4 times higher than that of CGL design demonstrating the reduction of the locking overhead as discussed in Sect. 5.1.

Effects of the merge operation Figure 22 shows aggregation throughput, while the size of the datasets having Uniform key distribution is increased. The parallel MTP aggregation step has almost constant throughput of about 820 MTuples/sec, which drops on very high cardinalities due to the usage of a small Lock CAM as explained in 5.3.1. The effect of a small Lock CAM size is less pronounced on larger datasets as the engines spend more time in the steady state. The merge step introduces an overhead; however, it comes at a fixed price. This cost depends solely on the key cardinality because aggregation reduces the initial input into a constant number of streams which should be merged. Hence, as the size of the relation grows the merge step overhead gets amortized.

FPGA resources utilization Table 3 shows the resource utilization (registers, LUTs, and BRAMs used) for the two MTP aggregation designs (multiplexed, FGL) as the number of engines is scaled up. The biggest drivers of resource usage in these engines are the CAMs. The CAMs are the largest components in the engines and dictate size and timing constraints. It is interesting to note that the 8 engine FGL design is comparable to the previous design, showing that the increased complexity of the lower level locks is not too complex to implement in hardware. We were also able to save significantly in BRAM usage as well. The aggregation design uses only 62% of the available resources showing there is still room to incorporate other relational operations on the FPGA fabric.

6 Related work

Acceleration of database operations Commercial in-memory database systems include SAP HANA [23] which relies on multi-core CPUs, large main memory and caches as well as data compression. IBM BLU [8] attempts to keep data on the processor and its caches so as to reduce DRAM access as much as possible. MS SQL Hekaton [21] provides a lock-free data structure to provide high level of concurrency. Oracle's TimesTen [40] relies heavily on caches hierarchies. There are also academic research prototypes such as Peloton [47] which provides autonomous tuning for relational databases. GPU-based solutions have also been proposed such as [31], while other are a hybrid of CPU and FPGA architectures have been proposed to mitigate long memory latencies [52].

Many recent works have considered the in-memory implementation of join and aggregation relational operators (hash- or sort-based). Sort-merge joins on modern CPUs were initially considered by Kim et al. [35]. This implementation explored the use of SIMD operations and hypothesized that sort-merge join performance will surpass the hash-based algorithms, given wider SIMD registers. Subsequent work [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. Balkesen et al. [5] reconsidered the issue and found that hash joins still have an edge over sort-merge implementations even with the latest advance in width of SIMD registers and NUMA-aware algorithms. There has been a recent interest in tuning the join for specialized processing units such as Intel Xeon Phi [15]. PolyHJ [34] proposes a hybrid hash join paradigm that can dynamically execute different hash join models and tackle size skew. The proposed algorithm adapts behavior based on input relation and hardware characteristics.

In addition to joins, group-by aggregation operator, relying on multi-threaded architectures to boost its performance, was also extensively researched. One of the earliest works [16] explores different aggregation implementations on chip multiprocessors (CMPs) and concludes that performance largely depends on input characteristics like key cardinality, thus opting for adaptive strategy based on sampling.

Follow-up work from the same authors [17] specifically explores the partitioning step of hash aggregation in the same CMP environment and, in line with [43], emphasizes the thread coordination as a key component of this step. The work by Ye et al. [59] considers both partitioning-based and non-partitioned aggregation implementations and proposes several hybrid approaches, which outperform previous implementations on Intel Nehalem architecture. In this work, we focus mainly on non-partitioned versions of algorithms. Wang et al. [56] describes novel NUMA-aware partitioned in-memory hash aggregation algorithm, which avoids cache coherency misses and minimizes locking costs. Finally, more recent work has shown the improvements of using novel CPU hardware. Cheng et al. [15] demonstrated a highly parallel in-memory join on the 64-core Intel Knight's Landing platform. Pohl et al. [48] have shown how HBM (high bandwidth memory) can be used as another layer in the storage hierarchy to improve performance. Hash- and sort-based aggregation is evaluated in [46]. The findings show that both approaches have the same complexity in terms of cache line transfers. An adaptive algorithmic framework based on sorting by hash value enables switching between hashing and sorting approaches during run-time based on a criterion of locality. The framework is cache-efficient and can be tuned depending on the hardware.

While the software community has examined both hash and sort-merge for join and aggregation operators, the FPGA community has concentrated on sort-merge approaches. The reasons for this are twofold. Firstly, sorting and merging implementations are easily parallelized on FPGA architectures. For example, sorting networks like bitonic-merge [32] and odd-even sort [39] are well-established designs for FPGAs; Casper et al. [13] developed a multi-FPGA sort-merge algorithm, while other works [50,58] used sort-merge as part of a hardware database processing system. Secondly, building an in-memory hash table efficiently is non-trivial task because of the required synchronization.

An FPGA-accelerated implementation of group-by aggregation was first considered by Mueller et al. [45]. This work also utilized CAMs in the implementation of the aggregation operator, but in a very narrow scope, i.e., using CAMs to match an incoming tuple with the appropriate group. Hence, the work continued long tradition of using CAMs to answer set-membership queries (previously explored in applications like click-fraud, online intrusion detection [7]). Our design also uses CAMs, but is different from previous approaches in two ways: (i) In addition to the key, we store and update the aggregate value locally in the CAM, and (ii) we use CAMs as a synchronization primitive to resolve conflicts during updates.

Recently, we used the MTP execution to accelerate the selection operator [11] by masking long memory latencies and managing thousands of threads concurrently without using any caches, as opposed to software CPU-based implementations, which require effective caching to limit memory requests. Using the MTP approach to implement a given database operator requires a design specific to the operator's characteristics. For example, the selection operator applies the query predicate to all the tuples in a relation. This operation is thus *partitionable*, meaning that each tuple could be processed independently of the others. Hence, checking a predicate on a given tuple is an independent thread [11]; as a result, there is no need for inter-thread synchronization

which is an important characteristic for both hash-join and group-by aggregation.

Kim et al. proposed BionicDB [36], an FPGA accelerator for OLTP workloads. Similar to our fine-grained locks, they used locks (implemented in BRAM) on the towers of jump-lists to improve throughput. Ma et al. [42] demonstrate a similar multithreaded FPGA model in the area of graph workloads.

Content addressable memories A CAM is a memory where every bit is paired with a comparator allowing concurrent matching to a search word. Its ability to perform a search in unit time comes at a high cost of area, energy, and long clock cycle time.

As the number of entries in the CAM increases, the achievable clock frequency of the circuit drops. This limitation either restricts the size of the CAM or increases the number of cycles it takes to perform a search or an update operation. Nonetheless, CAMs have proven to be very useful in domains such as networking (e.g., implementing an IP table in a network router). Recently, we explored how CAMs can be used to accelerate the breadth first search algorithm [57].

In a streaming environment, CAMs can maintain a cache of recently seen unique items and allow quick access to them without stalling the pipeline. This fast cache look-up mechanism can also be used as a fine-grained address-based synchronization primitive, which avoids long latency trips to main memory and does not require special hardware. Consider the case when a CAM is assigned to guard a particular memory partition. It can be configured to hold the addresses of the values that need synchronized access. If all memory requests within a partition are first submitted to the CAM, before being routed to the memory, the accesses to identical addresses are serialized locally in the CAM. In this case, a CAM entry serves as an exclusive lock, which gets released (flushed from the CAM) after the request(s) completion. In [1], we discuss how to use this approach for synchronization in the multithreading group-by aggregation algorithm.

To the best of our knowledge, all previous FPGA implementations relied on specialized platform features to provide synchronization primitives. In our previous work [28], we used atomic operations provided by the now discontinued Convey MX architecture [18]. Each word in memory maintains a locking bit that can be set by a specialized test-and-set memory instruction. Leveraging CAMs for synchronization increases the portability of our design by moving all synchronization operations to the FPGA. In addition, this design provides more selective fine-grained synchronization primitives in comparison with the Convey-MX, which places a lock on all FPGA-memory communication channels.

It was shown that implementing fully associative matching logic for CAMs on both Altera and Xilinx FPGAs introduces a 60× space overhead compared to regular BRAMs [60]. Dhawan et al. [20] explored various designs of CAMs and introduced a trade-off between CAM size and update time.

7 Conclusion

In this paper, we implemented and evaluated hash-join and hash-based group-by aggregation, using hardware multithreading techniques on FPGA hardware accelerators. The data structures are kept in global memory, which increases the access latency compared to on-chip BRAMs, but allows us to tackle much larger problem sizes. Multithreading allows the MTP design to mask the longer latency by issuing hundreds of threads across four FPGAs.

A hash join design for key/value store databases has been presented. Throughput experiments over three datasets, ranging from uniform to slightly skewed, showed that the MTP design outperforms the best currently available software implementations by 2× to 3.4×. However, on extremely skewed datasets the MTP design's performance suffered compared to software approaches.

Our hash join design was also extended to support wider tuples, and larger key sizes. We tested out implementation on join queries from the TPC-H benchmark suite. Throughput results dropped compared to the key/value design for both the MTP design and software. However, the MTP design outperformed the non-partitioned CPU results by 1.3× to 1.5×. The MTP design outperformed the partitioned CPU results by over 4×.

The same multithreading techniques are also used to implement a group-by aggregation function on the two MTP designs. Aggregation is a more complex operation because threads can either update an existing node or create a new node. Compared to hash join, where every thread in the build relation creates a new node, every thread in the probe relation only reads the data structures. We evaluate the FGL MTP design against five software approaches (both partitioned and non-partitioned) over five different datasets. Experiments show a sharp decline in performance for the software approaches as the cardinality increases. The FGL MTP design's throughput is unaffected by the benchmark's cardinality and can sustain between 500 and 1500 MTuples/sec depending on the key distribution, achieving an average of 3.3× speedup over all CPU implementations. Due to the limited clock frequency, memory bandwidth, and limited on-chip memory of the FPGA platform relative to state-of-the-art multi-core CPUs, we cannot demonstrate superior raw throughput. However, this proof-of-concept work demonstrates throughput improvements achieved by efficient memory bandwidth utilization using latency-masking threads.

Acknowledgements This research was partially supported by NSF Grants: IIS-1838222, IIS-1527984, IIS-1447826 and IIS-1305253.

References

1. Absalyamov, I., Budhkar, P., Windh, S., Halstead, R.J., Najjar, W.A., Tsotras, V.J.: FPGA-accelerated group-by aggregation using synchronizing caches. In: Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN '16, pp. 11:1–11:9. NY, USA: ACM (2016)
2. Albutiu, M.-C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. Proc. VLDB Endow. **5**(10), 1064–1075 (2012)
3. Alpha Data. http://www.alpha-data.com/dcp/capi.php (2015)
4. AWS Events. AWS re:invent 2019: Amazon redshift reimagined: RA3 and AQUA (ANT230). https://youtu.be/6pZrE_tveLI (2019). Accessed 2020-7-11
5. Balkesen, C., Alonso, G., Teubner, J., Özsu, M .T.: Multi-core, main-memory joins: sort vs. Hash revisited. Proc. VLDB Endow. **7**(1), 85–96 (2013)
6. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.: Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. In: Proceedings of the 2013 IEEE International Conference on Data Engineering, ICDE'13, pp. 362–373 (2013)
7. Bandi, N., Metwally, A., Agrawal, D., El Abbadi, A.: Fast data stream algorithms using associative memories. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, pp. 247–256 (2007)
8. Barber, R., Lohman, G., Raman, V., Sidle, R., Lightstone, S., Schiefer, B.: In-memory BLU acceleration in IBM's DB2 and dashDB: optimized for modern workloads and hardware architectures. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 1246–1252. ieeexplore.ieee.org (2015)
9. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory Hash join algorithms for multi-core CPUs. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD'11, pp. 37–48 (2011)
10. Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: memory access. In: Proceedings of the 25th International Conference on Very Large Data Bases, VLDB'99, pp. 54–65 (1999)
11. Budhkar, P., Absalyamov, I., Zois, V., Windh, S., Najjar, W.A., Tsotras, V.J.: Accelerating in-memory database selections using latency masking hardware threads. ACM Trans. Archit. Code Optim. **16**(2), 13:1–13:28 (2019)
12. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 141–151 (2012)
13. Casper, J., Olukotun, K.: Hardware acceleration of database operations. In: Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 151–160 (2014)
14. Chen, S., Ailamaki, A., Gibbons, P.B., Mowry, T.C.: Improving hash join performance through prefetching. ACM Trans. Database Syst. **3**, 32–3 (2007)
15. Cheng, X., He, B., Du, X., Lau, C.T.: A study of main-memory hash joins on many-core processor: a case with intel knights landing architecture. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM' 17, pp. 657–666. ACM, NY, USA (2017)
16. Cieslewicz, J., Ross, K.A.: Adaptive aggregation on chip multiprocessors. In: International Conference on Very Large Data Bases, VLDB '07, pp. 339–350 (2007)
17. Cieslewicz, J., Ross, K.A.: Data partitioning on chip multiprocessors. In: Proceedings of the 4th International Workshop on Data Management on New Hardware, DaMoN '08, pp. 25–34 (2008)
18. Convey Computers. http://www.conveycomputer.com (2015)
19. (dbInsight), T.B.: Amazon redshift turns AQUA. https://www.zdnet.com/article/amazon-redshift-turns-aqua/. Accessed 2020-8-17
20. Dhawan, U., Dehon, A.: Area-efficient near-associative memories on FPGAs. ACM Trans. Reconfigurable Technol. Syst.: TRETS **7**(4), 1–22 (2015)
21. Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server's memory-optimized OLTP engine. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD '13, pp. 1243–1254. ACM, NY, USA (2013)
22. Fang, J., Mulder, Y.T.B., Hidders, J., Lee, J., Hofstee, H.P.: In-memory database acceleration on FPGAs: a survey. VLDB J. **29**, 33–59 (2019)
23. Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., Dees, J.: The SAP HANA database—an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)
24. Fernandez, E.B., Najjar, W.A., Lonardi, S., Villarreal, J.: Multithreaded FPGA acceleration of DNA sequence mapping. In: 2012 IEEE Conference on High Performance Extreme Computing, pp. 1–6. ieeexplore.ieee.org (2012)
25. Francisco, P.: The Netezza data appliance architecture: a platform for high performance data warehousing and analytics. IBM Redbook. REDP-4725-00 (2011). https://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf
26. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.: Quickly generating billion-record synthetic databases. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, SIGMOD'94, pp. 243–252 (1994)
27. Gupta, P.K.: Accelerating datacenter workloads. In: 26th International Conference on Field Programmable Logic and Applications (FPL). fpl2016.org (2016)
28. Halstead, R.J., Absalyamov, I., Najjar, W.A., Tsotras, V.J.: FPGA-based multithreading for in-memory hash joins. In: CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4–7, 2015, Online Proceedings, CIDR'15 (2015)
29. Halstead, R.J., Najjar, W.A., Huseini, O.: SpVM acceleration with latency masking threads on FPGAs. Algorithms **20**, 21 (2014)
30. Halstead, R.J., Sukhwani, B., Min, H., Thoennes, M., Dube, P., Asaad, S., Iyer, B.: Accelerating join operation for relational databases with FPGAs. In: Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM'13, pp. 17–20 (2013)
31. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. **34**(4), 21:1–21:39 (2009)
32. Ionescu, M., Schauser, K.: Optimizing parallel bitonic sort. In: Proceedings of the 11th International Symposium on Parallel Processing, pp. 303–309 (1997)

33. Jun, S., Xu, S.: Terabyte sort on FPGA-accelerated flash storage. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 17–24. ieeexplore.ieee.org (2017)
34. Khattab, O., Hammoud, M., Shekfeh, O.: Polyhj: a polymorphic main-memory hash join paradigm for multi-core machines. In: Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM' 18, pp. 1323–1332. ACM, NY, USA (2018)
35. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. Proc. VLDB Endow. **2**(2), 1378–1389 (2009)
36. Kim, K., Johnson, R., Pandis, I.: BionicDB: fast and power-efficient OLTP on FPGA. In: EDBT (2019)
37. Krikelis, A., Weems, C.C.: Associative processing and processors. Computer **27**(11), 12–17 (1994)
38. Kuehn, J.T., Smith, B.J.: The horizon supercomputing system: architecture and software. In: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Supercomputing '88, pp. 28–34. IEEE, Los Alamitos, CA, USA (1988)
39. Kumar, M., Hirschberg, D.: An efficient implementation of batcher's odd-even merge algorithm and its application in parallel sorting schemes. IEEE Trans. Comput. **100**(3), 254–264 (1983)
40. Lahiri, T., Neimat, M.-A., Folkman, S.: Oracle TimesTen: an in-memory database for enterprise applications. IEEE Data Eng. Bull. **36**(2), 6–13 (2013)
41. Lee, J., Kim, H., Yoo, S., Choi, K., Hofstee, H.P., Nam, G.-J., Nutter, M.R., Jamsek, D.: Extrav: boosting graph processing near storage with a coherent accelerator. Proc. VLDB Endow. **10**(12), 1706–1717 (2017)
42. Ma, X., Zhang, D., Chiou, D.: FPGA-accelerated transactional execution of graph workloads. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'17, pp. 227–236. ACM, NY, USA (2017)
43. Manegold, S., Boncz, P., Kersten, M.: Optimizing main-memory join on modern hardware. IEEE Trans. Knowl. Data Eng. **14**(4), 709–730 (2002)
44. MemSQL. https://www.memsql.com (2013)
45. Mueller, R., Teubner, J., Alonso, G.: Streams on wires: a query compiler for FPGAs. Proc. VLDB Endow. **2**(1), 229–240 (2009)
46. Müller, I., Sanders, P., Lacurie, A., Lehner, W., Färber, F.: Cache-efficient aggregation: hashing is sorting. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, pp. 1123–1136. ACM, NY, USA (2015)
47. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T.C., Perron, M., Quah, I.: Others. Self-driving database management systems. In: CIDR, vol. 4, p. 1. cc.gatech.edu (2017)
48. Pohl, C., Sattler, K.-U., Graefe, G.: Joins on high-bandwidth memory: a new level in the memory hierarchy. VLDB J. **29**, 797–817 (2020)
49. Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P., Burger, D.: A reconfigurable fabric for accelerating large-scale datacenter services. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), ISCA'14, pp. 13–24 (2014)
50. Sadoghi, M., Javed, R., Tarafdar, N., Singh, H., Palaniappan, R., Jacobsen, H.-A.: Multi-query stream processing on FPGAs. In: Proceedings of the 2012 IEEE International Conference on Data Engineering, ICDE'12, pp. 1229–1232 (2012)
51. Sheffield, D.: IvyTown Xeon+ FPGA: the HARP program. In: International Symposium on Computer Architecture (ISCA): Tutorial (2016)
52. Sidler, D., Istvan, Z., Owaida, M., Kara, K., Alonso, G.: doppioDB: a hardware accelerated database. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, pp. 1659–1662. ACM, NY, USA (2017)
53. Thistle, M.R., Smith, B.J.: A processor architecture for Horizon. In: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, pp. 35–41 (1988)
54. Tözün, P., Gold, B., Ailamaki, A.: OLTP in wonderland: where do cache misses come from in major OLTP components? In: Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN '13, pp. 8:1–8:6. ACM, NY, USA (2013)
55. Vahidsafa, A., Turullols, S., Smentek, D., Sivaramakrishnan, R., Loewenstein, P., Jairath, S., Feehrer, J.: The Oracle Sparc T5 16-core processor scales to eight sockets. IEEE Micro **33**, 48–57 (2013)
56. Wang, L., Zhou, M., Zhang, Z., Shan, M.-C., Zhou, A.: NUMA-aware scalable and efficient in-memory aggregation on large domains. IEEE Trans. Knowl. Data Eng. **27**(4), 1071–1084 (2015)
57. Windh, S., Budhkar, P., Najjar, W.A.: CAMs as synchronizing caches for multithreaded irregular applications on FPGAs. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD '15, pp. 331–336. IEEE, Piscataway, NJ, USA (2015)
58. Wu, L., Lottarini, A., Paine, T.K., Kim, M.A., Ross, K.A.: Q100: the architecture and design of a database processing unit. In: Proceeding of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pp. 255–268 (2014)
59. Ye, Y., Ross, K.A., Vesdapunt, N.: Scalable aggregation on multicore processors. In: Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN '11, pp. 1–9 (2011)
60. Yiannacouras, P., Rose, J.: A parameterized automatic cache generator for FPGAs. In: Proceedings IEEE International Conference on Field-Programmable Technology (FPT), pp. 324–327 (2003)
61. Zynq, X. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (2015)