FPGA-Accelerated Group-by Aggregation Using Synchronizing Caches

Ildar Absalyamov
iabsa001@cs.ucr.edu

Prerna Budhkar
pbudh001@cs.ucr.edu

Skyler Windh
windhs@cs.ucr.edu

Robert J. Halstead
rhalstea@cs.ucr.edu

Walid A. Najjar
najjar@cs.ucr.edu

Vassilis J. Tsotras
tsotras@cs.ucr.edu

Department of Computer Science & Engineering
University of California, Riverside

ABSTRACT

Recent trends in hardware have dramatically dropped the price of RAM and shifted focus from systems operating on disk-resident data to in-memory solutions. In this environment high memory access latency, also known as memory wall, becomes the biggest data processing bottleneck. Traditional CPU-based architectures solved this problem by introducing large cache hierarchies. However algorithms which experience poor locality can limit the benefits of caching. In turn, hardware multithreading provides a generic solution that does not rely on algorithm-specific locality properties.

In this paper we present an FPGA-accelerated implementation of in-memory group-by hash aggregation. Our design relies on hardware multithreading to efficiently mask long memory access latency by implementing a custom operation datapath on FPGA. We propose using CAMs (Content Addressable Memories) as a mechanism of synchronization and local pre-aggregation. To the best of our knowledge this is the first work, which uses CAMs as a synchronizing cache. We evaluate aggregation throughput against the state-of-the-art multithreaded software implementations and demonstrate that the FPGA-accelerated approach significantly outperforms them on large grouping key cardinalities and yields speedup up to 10x.

Keywords

FPGA; Hash Aggregation; Main Memory; Hardware Acceleration

1. INTRODUCTION

The rapidly decreasing cost of RAM has created a niche for in-memory analytics solutions. Fairly large datasets can now be stored and processed entirely in memory. One of the crucial operators in any OLAP query is the group-by aggregation, since its run-time makes up a large portion of the total query evaluation time. Thus, an efficient optimized implementation of group-by aggregation could significantly boost overall performance of analytical workloads. Among two possible implementations of aggregation algorithms, hash-based and sort-based, the former is generally preferred because it avoids the high penalty of sorting the input relation. Hence, in this paper we concentrate on an in-memory hash-based implementation for group-by aggregation.

While memory capacity continues to increase, the past decade has seen a stagnation of processor clock speeds caused by the end of the Dennard scaling. This leaves parallelism as the only option to allow fast processing for the growing amounts of memory-resident data. The computer architecture community considered two approaches to leverage parallelism, namely (i) off-the-shelf multi-core architectures, including CPUs and GPUs, [2, 10] or (ii) customizable architectures such as CPUs with FPGAs [14, 18, 13, 16, 21, 20]. While multi-cores typically have much higher clock speeds, specialized hardware (e.g., FPGA) has both the advantages of customization (the hardware design is optimized for a specific application) and parallelism. In custom architectures parallelism is usually achieved by replicating compute circuits, which could be accomplished giving very large size of modern FPGAs.

The major issue limiting performance of in-memory algorithms is the growing gap between the memory bandwidth and the speed of the processing unit (the so-called *memory wall*), which is even more important for multi-cores given their higher clock speeds. The multi-core approach addressed this problem by introducing large *cache hierarchies*, relying on the data locality (spatial and/or temporal) to mitigate memory latency. This solution does not come for free: cache hierarchies can take up to 80% of the chip area thus are becoming a limiting factor on the number of cores that can be accommodated on a single chip. Because of leakage current they also become a primary consumer of energy on the chip. Besides that, the extensive use of hashing renders multi-core implementations of group-by aggregation inefficient, since they do not exhibit any form of locality.

Rather than relying on a cache hierarchy, *hardware multithreading* aims to completely mask memory latency. In this execution model a running thread relinquishes execution to a ready thread, as soon as it performs a long-latency operation. The executing thread is then suspended until the long-latency operation completes and eventually returns to

DaMoN'16, June 27, 2016, San Francisco, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3638-3/15/06...$15.00

DOI: http://dx.doi.org/10.1145/2933349.2933360

a ready state again. This approach has been used in multi-cores (UltraSparc [10]). However these architectures support a relatively small number of threads because the CPU has to provision a full hardware context for each ready/waiting thread, thereby limiting the amount of parallelism.

In a custom architecture (e.g., FPGA) where the datapath is designed for a small number of predefined operations, the required context for each thread is much smaller than in a general-purpose CPU and hence more threads can be supported. In this multithreaded model the parallelism is limited only by the number of active threads (ready, executing or waiting). We have recently applied this multithreading approach to implement an in-memory hash join algorithm [12]. Our results demonstrated up to 10x higher throughput over the best multi-core software alternatives with comparable memory bandwidth.

In this paper we extend this idea to implement multithreaded in-memory hash-based group-by aggregation. Despite the seeming similarity, the two operators are using the hash table in a very different manner: the hash join has a clear delineation between the build phase, when the hash table is modified, and the probe phase during which the table is only read. In the group-by aggregation the read- and write-requests are instead mixed in a single phase. Moreover during the build phase of the hash join, a key will always create a new node in the appropriate bucket list (assuming the classical implementation, where each hash table bucket is associated with a linked list). For the aggregation, a key is first searched within the appropriate bucket list and then it either updates an entry value (if this key has been found) or inserts a new entry into bucket list. All these dissimilarities become especially important in the multithreaded environment, when explicit synchronization is needed to guarantee correctness, leading to different optimization strategies for the two hash-based operators.

In the previous work [12] we achieved synchronization during the join build phase by using atomic operations, which acquire special locks on individual memory locations, a unique property of the Convey-MX architecture [8]. Apart from being vendor-specific, this design has a high synchronization overhead when it is applied to the problem of group-by aggregation. Instead we consider a generic solution based on Content Addressable Memories (CAMs). We show that a CAM-based implementation allows not only to correctly synchronize contending requests, but also to do pre-aggregation, thus effectively serving as a *synchronizing cache*.

The following summarizes the contributions of the paper:

- We apply hardware multithreading to implement an in-memory hash-based group-by aggregation algorithm.
- We propose CAMs as a *synchronizing cache* mechanism and demonstrate its efficiency for the hash aggregation operator.
- We evaluate the throughput of our hardware multithreaded implementation against known existing software algorithms and demonstrate speedup up to 10x for a wide range of workloads.

We proceed with related work described in Section 2. The CAM mechanism is introduced in Section 3 while the hardware implementation of the group-by aggregation is detailed in Section 4. Experimental results appear in Section 5 and conclusions in Section 6.

2. RELATED WORK

The large amount of relatively cheap DRAM memory in modern commodity servers has reignited interest in memory-optimized algorithms both in industry [19] and academia [4, 2]. In multi-core CPU architectures two main alternatives have been considered. The *hardware-conscious* algorithms are tightly tailored to the underlying hardware and perform preliminary data partitioning to reduce cache misses. Instead, the *hardware-oblivious* solutions try to mask latency by relying on hardware-provided multithreading. These contrasting approaches were extensively studied in the context of in-memory hash joins [4, 2] as well as sort-merge joins [1, 15].

Hardware-oblivious implementations of the group-by aggregation were explored by Cieslewicz et al. [6], who showed that performance largely depends on input characteristics (key cardinality). Follow up work [7] explored the partitioning step of hash aggregation and concluded that the thread coordination is a key component influencing the performance of this step. Finally, Ye et al. [23] proposed hybrid algorithms and showed that they outperform pure hardware-conscious and -oblivious implementations.

An FPGA-accelerated implementation of group-by aggregation was first considered by Mueller et al. [17]. 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 answering set-membership queries (previously explored in applications like click-fraud, online intrusion detection [3]). 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.

It was shown that implementing fully-associative matching logic for CAMs on both Altera and Xilinx FPGAs introduces a 60x overhead compared to regular BRAMs [24]. This drawback makes implementing large CAMs on reconfigurable fabrics notoriously hard. Dhawan et al. [9] explored various designs of CAMs and introduced a trade-off between CAM size and update time.

3. USING CAMS 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)

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 [22]. 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 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 Section 4.1 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 [12] we have used atomic operations which were implemented using locks on individual memory locations, provided by the now discontinued Convey MX architecture [8]. 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 with a sufficient area FPGA has direct access to the memory. In addition this design provides more selective fine-grained synchronization primitives in comparison to the Convey-MX, which places a lock on all FPGA-memory communication channels.

4. GROUP-BY AGGREGATION ON FPGA

In the rest we assume that the input relation fits in main memory but is too large to fit locally on the FPGA's memory. To fully utilize the memory bandwidth available to the FPGA we employ a hardware multithreaded model, which allows the FPGA to process ready jobs while idle jobs wait on (long) memory accesses. In this model the FPGA maintains a queue of *ready threads* that can be accessed in a single clock cycle. Whenever a thread issues a memory request the FPGA saves the thread state into local memory and picks up the next ready job. Once a memory request is fulfilled the thread state is updated, and queued back into the *ready threads* FIFO. If the FPGA can maintain more thread states than the memory latency then full latency masking is achieved, thus the bandwidth is fully utilized.

The mixed read-write nature of aggregation in conjunction with multiple outstanding requests requires us to use explicit synchronization to ensure correctness. Using atomic operations is one option, but this approach severely impacts the performance. Moreover, unlike the join operator, aggregated tuples exhibit temporal locality. We propose a novel multithreaded aggregation implementation based on CAMs. The design leverages explicit synchronization combined with the cache-like properties of the CAM. This fits perfectly in the context of group-by aggregation: firstly, the latency of a single aggregation job is hundreds of cycles, which means many interleaved jobs can have identical keys. With a CAM we can merge these jobs pre-aggregating the result locally on the FPGA and reduce the number of outstanding memory requests. This merging is achieved by leveraging cache properties of the CAM (allowing us to hold the aggregate value for a particular key). It also allows up 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 channels, therefore decrease granularity of the locks and boost the performance.

4.1 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 FPGA thread (job) and starts its pipelined execution. Figure 1 shows the state diagram for a single thread inside the aggregation engine. The *Filter CAM* is used to merge jobs with identical keys, hence reduces the memory request contention and minimizes the synchronization overhead. However due to hash collisions the synchronization cannot be avoided completely; thus the *Lock CAM* is used to acquire locks on hash table bucket

Table 1 shows an example of events and contents of *Filter* CAM, *Lock* CAM and main memory HashTable, while the input stream consists of 5 tuples with the following keys: A, C, A, B, A. The design assumes the COUNT aggregation function, 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 HashTable accesses (e.g., search, entry update, entry insert) take several cycles to finish. For example, *Job 1* sends a request to search value A in a hash table and gets response only at $Cycle_4$. *Lock CAM* maintains the locks for all buckets which are currently being searched or modified. In particular, after the job obtains a lock, it starts the bucket list search process and subsequently either updates an aggregate value or inserts a new entry into the bucket list for a certain key. Once a job completes, it invalidates the record in both CAMs, therefore frees up resources for other jobs. Jobs, 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 job waits until the lock is released, e.g. *Job 2* resumes its work only at $Cycle_5$. *Job 3* provides an example of early termination, because its value was locally aggregated in *Filter CAM*.

4.2 FPGA Design Optimizations & Tradeoffs

The main bottleneck of our design is memory bandwidth. In this paper we use a Convey-HC-2ex machine, but our designs are platform independent. In the Convey 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 initial design of our aggregation engine requires 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 Convey-HC-2ex has 16 memory channels, we replicate 4 engines ($\frac{16}{4}$) on a single FPGA thus leveraging inter-engine parallelism.

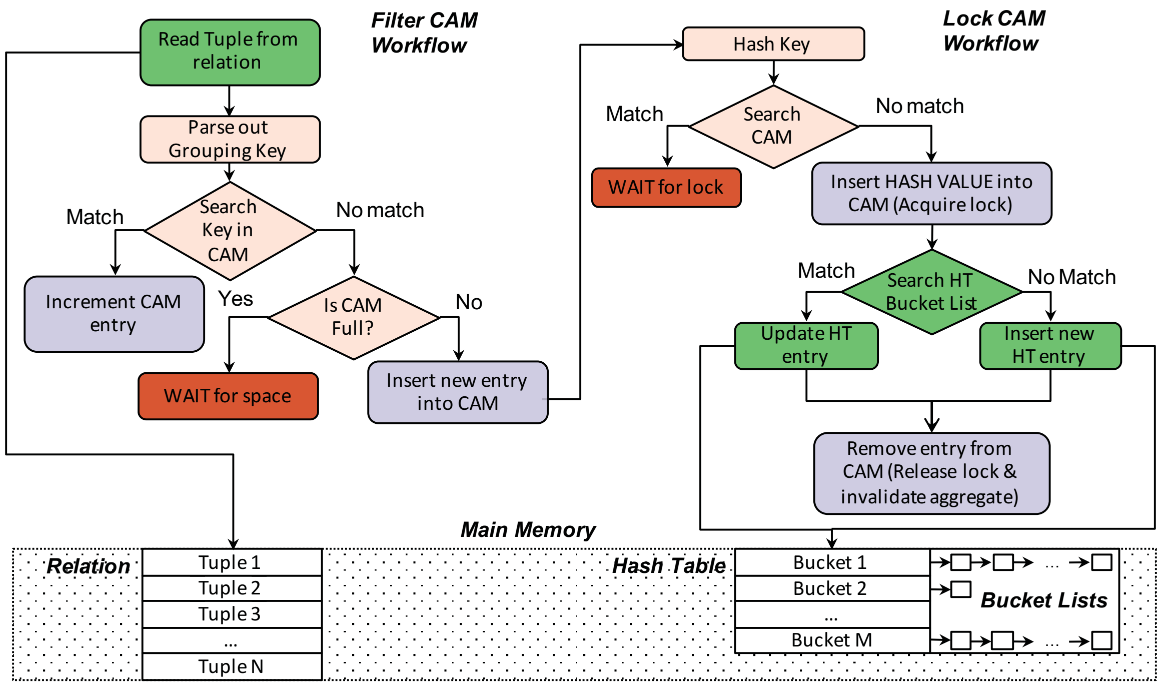


Figure 1: A state diagram for jobs in the aggregation engine.

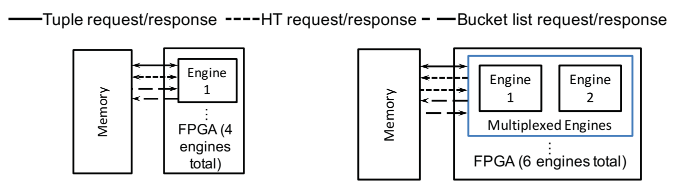
Cycle	Key	Filter CAM	Lock CAM	HashTable	Comments
1	A	*Miss, Insert (A,1)* {(A,1)}	*Miss, Insert hash(A)* {hash(A)}	{}	*Bucket*$_{hash(A)}$ is locked Request to search key A in HT is sent
2	C	*Miss, Insert (C,1)* {(A,1), (C,1)}	*Hit, since hash(A)=hash(C)* {hash(A)}	{}	*Job 2 waits for the lock*
3	A	*Hit, Update (A,2)* {(A,2), (C,1)}	{hash(A)}	{}	*Job 3 is discarded*
4		*Job 1 removes entry for key A* {(C,1)}	*Job 1 releases lock on hash(A)* {}	{(A,2)}	Key A was not found in HT Create new entry (A,2) in HT
5		{(C,1)}	*Job 2 obtains lock on hash(C)* {hash(C)}	{(A,2)}	*Bucket*$_{hash(C)}$ is locked Request to search key C in HT is sent
6	B	*Miss, Insert (B,1)* {(B,1), (C,1)}	*Miss, Insert hash(B)* {hash(C), hash(B)}	{(A,2)}	*Bucket*$_{hash(B)}$ is locked Request to search key B in HT is sent
7		*Job 2 removes entry for key C* {(B,1)}	*Job 2 releases lock on hash(C)* {hash(B)}	{(C,1), (A,2)}	Key A was not found in HT Create new entry (C,1) in HT
8	A	*Miss, Insert (A,1)* {(B,1), (A,1)}	*Miss, Insert hash(A)* {hash(A), hash(B)}	{(C,1), (A,2)}	*Bucket*$_{hash(A)}$ is locked Request to search key A in HT is sent
9		*Job 6 removes entry for key B* {(A,1)}	*Job 6 releases lock on hash(B)* {hash(B)}	{(B,1), (C,1), (A,2)}	Key B was not found in HT Create new entry (B,1) in HT
10		*Job 8 removes entry for key A* {}	*Job 8 releases lock on hash(A)* {}	{(A,3), (B,1), (C,1)}	Key A was found in HT Update entry for the key A in HT to (A,3)

Table 1: Contents of the *Filter CAM*, *Lock CAM* and HashTable (HT) and *modifications* altering all of them, while relation with the following keys is processed: A, C, A, B, A. **Assume hash(A)=hash(C). Initially both CAMs are empty. *Filter CAM* maintains the occurrence of duplicate keys, while *Lock CAM* locks the hash bucket, holding the bucket list's head pointer**

Figure 2(a) demonstrates the design and channel assignment of the replicated engine approach. Each replicated engine uses its own CAM for synchronization. As a result, values are aggregated in separate hash tables. However, this requires an extra merging phase at the end of the computation, an overhead which grows as we increase the number of engines per FPGA.

In addition to inter-engine parallelism we also improve intra-engine channel usage. Our initial 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. job recycling through CAM synchronization) introduces stalls and decreases the effective throughput.

In order to increase memory utilization we have *multiplexed* a pair of engines on a same set of memory channels, thus allowing the same channel to be used by two different engines. This means that the following engine operations (e.g. send and receive tuple request and response, read and



(a) Replicated engine design (b) Multiplexed engine design

Figure 2: Alternative engine placement strategies on a single FPGA with 16 memory channels.

write respective values to the hash table, 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 previous design, the new multiplexed engine uses 5 memory channels (adding an extra channel for accessing the in-memory hash table). This allows us to place 6 engines ($2 * \lfloor \frac{16}{5} \rfloor$) on a single FPGA. Figure 2(b) shows how engines are multiplexed on a single FPGA and depicts channel allocation in this design.

5. EXPERIMENTAL RESULTS

We chose the Convey HC-2ex as our target FPGA platform because of its high bandwidth memory access. In particular, the memory system that interfaces to the FPGA allows up to 16 concurrent memory requests per cycle per FPGA. The FPGA aggregation implementation is compared in terms of overall throughput against the best multi-core approaches [6, 23] running on a single processor with 4 parallel threads. We proceed with a short description of the Convey HC-2ex, followed by a summary of the various software aggregation algorithms as well a description of the datasets used in the experiments.

5.1 Convey HC-2ex Platform

The Convey HC-2ex is a heterogeneous platform that offers a shared global memory space between the CPU and FPGA regions. As shown in Figure 3a 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. Each processor has a peak memory bandwidth of 51.2 GB/s.

The hardware region has 4 Xilinx Virtex6-760 FPGAs connected to the global memory through a full crossbar. Each FPGA has 8 64-bit memory controllers running at 300MHz (Figure 3b). The FPGA logic cells run in a separate 150 MHz clock domain to ease timing and are connected to the memory controllers through 16 channels. The hardware region has 64 GB of 1600 MHz DDR3 RAM. Each FPGA has a peak memory bandwidth of 19.2 GB/s.

To approach a fair comparison, we run our experiments on 2 FPGAs to match memory bandwidth as close as possible (38.4 GB/s for the FPGA vs 51.2 GB/s for the CPU).

5.2 Software Implementations

In order to evaluate our FPGA-based solution we have implemented the following state-of-the-art multithreaded software aggregation algorithms: (i) Independent Tables[6], (ii) Shared Table [6], (iii) Hybrid Aggregation [6], (iv) Partition with Local Aggregation Table [23] and (v) Partition & Aggregate [23]. Here, (i) and (ii) are considered as non-partitioned approaches, while (iii) and (iv) are hybrid, and (v) is a partitioned approach.

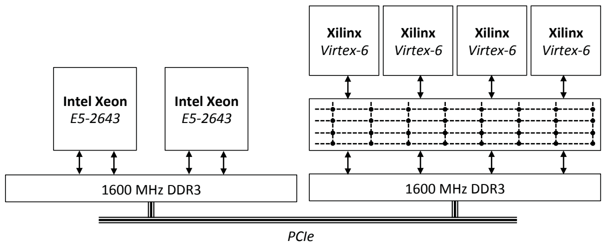
- **Independent Tables** [6] is the approach most similar to our hardware implementation. The tuples are evenly split among separate software threads (without partitioning), and each thread aggregates result into its own hash table. Once the aggregation is complete all tables are merged together, which requites write synchronization.

- **Shared Table (with locking or atomic synchronization)** [6] splits the tuples evenly between threads, but all threads aggregate their results into a single hash table, 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 don't have any difference from mutexes on medium and large cardinalities, so we choose atomics as a default synchronization primitive in all further experiments.

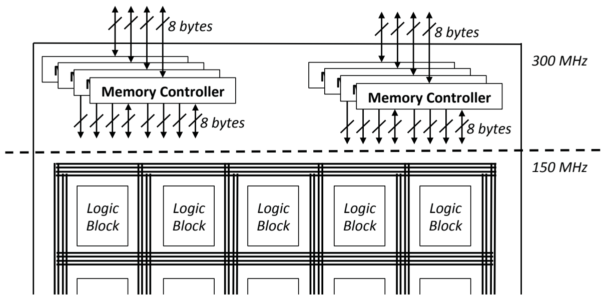
- **Hybrid Aggregation** [6] is a combination of two previous approaches. This algorithm allocates a small hash table for each thread. The size of the table is calculated based on the processor's L2 size to avoid cache misses. If the local table has enough space for a new value, or the value already 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 & Aggregate** [23] (also known as count-then-move [7]) 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, 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)** [23] 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 preprocessing 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. In the end all tables,



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



(b) Convey HC-2ex FPGA AE Wrapper

Figure 3: The Convey HC-2ex architecture is divided into software and hardware regions as shown in (a). Each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in (b)

which were produced during aggregation, are concatenated together, while local aggregation tables are synchronously merged in.

5.3 Dataset description

We use five datasets with various s key distributions, namely: Uniform, Heavy Hitter, Moving Cluster [6], Self Similar and Zipf_0.5.

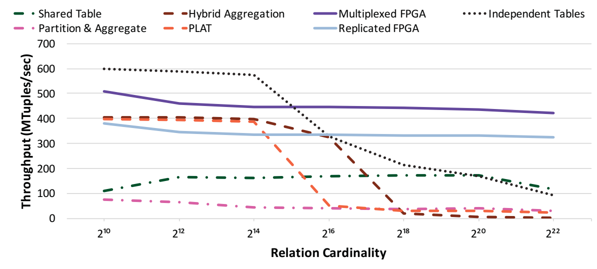
- In the **Uniform** dataset all key values are picked from *uint64* key range with uniform probability. After that generated key/value pairs are randomly shuffled.
- A half of the tuples in the **Heavy Hitter** dataset [6] share the same a key value. The remaining key values are picked uniformly and evenly distributed throughout the the entire relation.
- In the **Moving Cluster** dataset [6] 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 are 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. [11].
- In the **Zipf** dataset key values follow the Zipf distribution with a skew coefficient of 0.5. The generation algorithm appears in aforementioned work[11].

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 [23]). Each dataset used the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query processing algorithms [1, 5, 4] 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). This could be easily supported by adding a key extraction component into the FPGA 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.

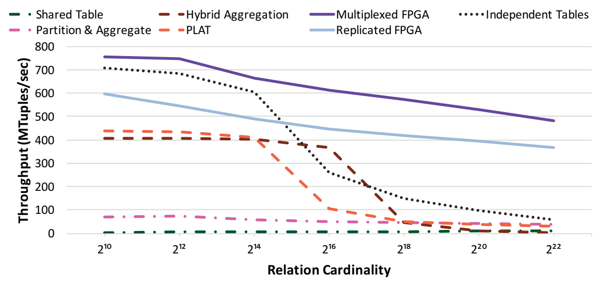
5.4 Throughput Evaluation

Figure 4 displays the throughput of the group-by aggregation as the key cardinality is increased, obtained for various datasets. Throughput was measured across two FPGA engine designs (regular and multiplexed), and five software (two non-partitioned, two hybrid and one partitioned) implementations. Throughput for skewed Heavy Hitter dataset Figure 4(d) resembles the results for Self Similar dataset Figure 4(b), while the throughput for moderately skewed data Zipf_0.5 4(e) is similar to the results obtained for Uniform dataset Figure 4(a). Software implementations demonstrate the best performance on Moving cluster dataset Figure 4(c) due to the property of the data distribution: similar grouping keys appear in the input stream clustered together, increasing CPU-cache hit rates.

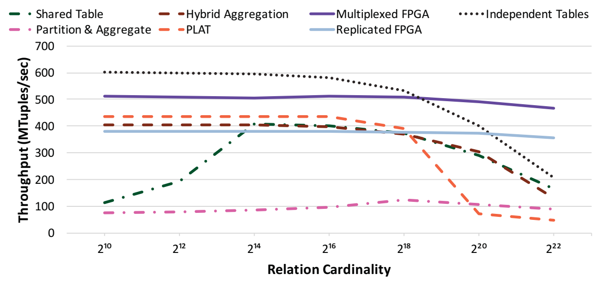
Despite all the differences in data distribution CPU aggregation performance mainly depends on the dataset's key cardinality. While the number of unique keys is low, hash tables can fit into the CPU cache entirely. However, as the cardinality increases, cache misses start to hamper the throughput due 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 & Aggregate) and non-partitioned (Shared Table)



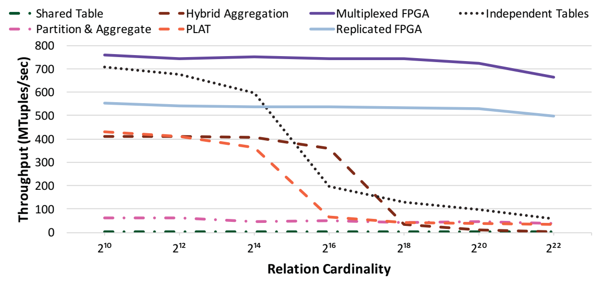
(a) Uniform



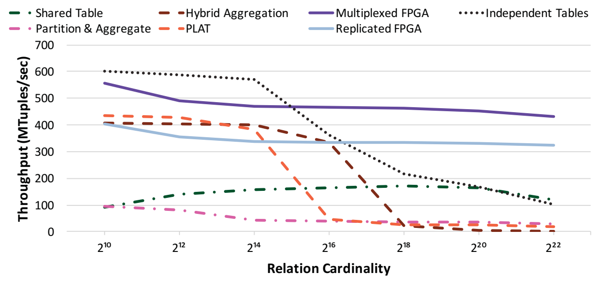
(b) Self Similar



(c) Moving Cluster



(d) Heavy Hitter



(e) Zipf 0.5

Figure 4: Aggregation throughput of hardware and software approaches for datasets with 256M tuples.

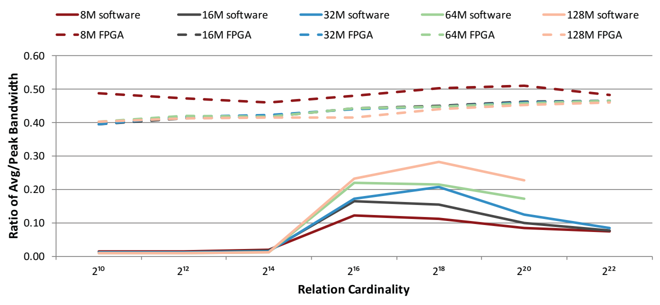


Figure 5: Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the Multiplexed FPGA design for varying dataset sizes and key cardinalities.

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 FPGA performance also drops as the key cardinality increases, however this effect is much less profound. Unlike the software throughput, this result is explained by the overhead, introduced by the post-processing merge step. However the overall performance is still up to 10x higher than the software throughput. The results also clearly show the benefits of the multiplexed engine design. Typically the throughput of the multiplexed FPGA engine is up to 30% more than the initial design. It should be also noted that the FPGA throughput does not deteriorate on heavily skewed data (*Self Similar*), as it was the case with the hash join [12].

Discussion: It should be noted that the performance benefits of the FPGA-based approaches come not from architecture-specific features, but from multithreading, which allows to utilize the available memory much better than any of the software implementations. Figure 5 depicts the ratio of effective average memory bandwidth to peak theoretical memory bandwidth for the best software (Independent Tables) and FPGA (multiplexed) implementations while varying dataset sizes and key cardinalities. Hardware mutithreading approach allows our FPGA implementation to keep the ratio almost constant, irrespectively of dataset size or key cardinality. On the contrary, the ratio for the software approach varies greatly. The effective memory bandwidth of the CPU implementation tends to grow as the size of the relation increases (from 8M to 128M), whereas the FPGA-based approach is less susceptible to data size variations. For low cardinality the aggregated relation and hash table are cached and there are almost no memory accesses, hence the ratio approaches 0. The software ratio peaks at around 0.3 for cardinality 2^{18}, but drops significantly for higher key cardinalities. For very large cardinalities the FPGA implementation ratio is almost 5 times higher.

5.5 Effects of the Merge Operation

The Figure 6 shows aggregation throughput while the size of the datasets having Uniform key distribution is increased. The parallel FPGA aggregation step has almost constant

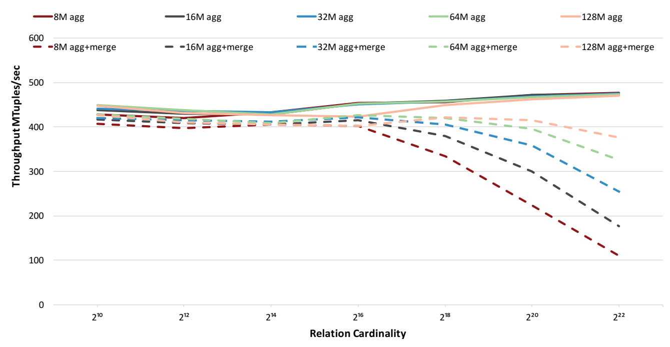


Figure 6: Effect of varying relation sizes on the FPGA aggregation throughput for datasets with Uniform key distribution. Solid lines represent throughput of the aggregation step (without merge operation), while dashed lines represent end-to-end (aggregation followed by the merge) throughput.

# of Engines	Registers	LUTs	BRAMs
1	99597 (11%)	87194 (18%)	126 (17%)
2	116635 (13%)	100497 (21%)	147 (19%)
3	135517 (15%)	115560 (24%)	184 (24%)
4	152132 (17%)	129775 (27%)	206 (28%)
1-Multiplexed	113695 (11%)	114280 (24%)	142 (19%)
2-Multiplexed	145690 (15%)	140684 (29%)	196 (27%)
3-Multiplexed	179641 (18%)	200175 (42%)	250 (34%)

Table 2: FPGA resource utilization for aggregation engines.

throughput of about 450 MTuples/sec, even on very high cardinalities. 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, so that the full throughput is almost constant for relations greater than 128 million tuples.

5.6 FPGA Area Utilization

Table 2 shows the resource utilization (registers, LUTs, and BRAMs used) for both FPGA aggregation designs (replicated and multiplexed) as the number of engines is scaled up. As we can see increasing the number of engines by one only adds an additional 2% for registers, 3% for LUTs, and 4% for BRAMs for replicated engine design. This happens because a lot of the components are shared across the engines. However as we start multiplexing the engines we stop sharing the resources due to timing constraints. This results in growth of FPGA resource utilization as we increase the number of engines. The aggregation design utilizes a lot of LUTs, which are extensively used in our CAM implementation. The hardened BRAM blocks only have two channels. This property is too restrictive for the CAMs, which must access all locations in parallel. The aggregation design uses only 42% of the available resources showing there is still room to incorporate other relational operations on the FPGA fabric.

6. CONCLUSIONS

In this paper we presented a multithreaded FPGA implementation of the group-by hash aggregation operation. We introduce a portable approach which uses CAMs to provide fast caching and enforce synchronization. We explore various FPGA designs and apply optimizations to further improve the performance. Experimental results show that the aggregation throughput is consistent and predictable regardless of a relation's size and cardinality. Despite the fact that the final merge step does affect performance, we show that this overhead is amortized when the relation size increases. Experiments show that the multithreaded FPGA approach can significantly outperform all existing software approaches and demonstrate especially good performance for high cardinality benchmarks. Throughput ranges between 700 to 150 MTuples/sec depending on the dataset distribution and key cardinality, with a speedup up to 10x.

7. ACKNOWLEDGMENTS

This work was partially supported by NSF grants IIS - 1161997 and CCF-1219180.

8. REFERENCES

[1] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. In *Proceedings of the VLDB Endowment*, VLDB'13, pages 85–96, 2013.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. Özsu. 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, pages 362–373, 2013.

[3] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi. Fast data stream algorithms using associative memories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'07, pages 247–256, 2007.

[4] S. Blanas, Y. Li, and J. M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 37–48, 2011.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the VLDB Endowment*, VLDB'99, pages 54–65, 1999.

[6] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the VLDB Endowment*, VLDB'07, pages 339–350, 2007.

[7] J. Cieslewicz and K. A. Ross. Data Partitioning on Chip Multiprocessors. In *Proceedings of the International Workshop on Data Management on New Hardware*, DaMoN'08, pages 25–34, 2008.

[8] Convey Computers. http://www.conveycomputer.com, 2015.

[9] U. Dhawan and A. Dehon. Area-Efficient Near-Associative Memories on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):30:1–30:22, Jan. 2015.

[10] J. Feehrer, S. Jairath, P. Loewenstein, et al. The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets. *IEEE Micro*, 33(2):48–57, March 2013.

[11] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly Generating Billion-record Synthetic Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'94, pages 243–252, 1994.

[12] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. FPGA-based Multithreading for In-Memory Hash Joins. In *Biennial Conference on Innovative Data Systems Research*, CIDR'15, 2015.

[13] IBM Netezza. www.ibm.com/software/data/netezza/, 2014.

[14] Kickfire. http://www.teradata.com/, 2015.

[15] C. Kim, T. Kaldewey, V. W. Lee, et al. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. In *Proceedings of the VLDB Endowment*, VLDB'09, pages 1378–1389, 2009.

[16] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *Biennial Conference on Innovative Data Systems Research*, CIDR'09, 2009.

[17] R. Mueller, J. Teubner, and G. Alonso. Streams on Wires: A Query Compiler for FPGAs. In *Proceedings of the VLDB Endowment*, VLDB'09, pages 229–240, 2009.

[18] A. Putnam, A. M. Caulfield, E. S. Chung, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Annual International Symposium on Computer Architecture*, ISCA'14, June 2014.

[19] V. Raman, G. Attaluri, R. Barber, et al. DB2 with BLU Acceleration: So Much More Than Just a Column Store. In *Proceedings of the VLDB Endowment*, VLDB'13, pages 1080–1091, August 2013.

[20] M. Sadoghi, R. Javed, N. Tarafdar, et al. Multi-query Stream Processing on FPGAs. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE'12, pages 1229–1232, 2012.

[21] J. Teubner and R. Mueller. How Soccer Players Would Do Stream Joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 625–636, 2011.

[22] S. Windh, P. Budhkar, and W. A. Najjar. CAMs as Synchronizing Caches for Multithreaded Irregular Applications on FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD'15, pages 331–336, 2015.

[23] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the International Workshop on Data Management on New Hardware*, DaMoN'11, pages 1–9, 2011.

[24] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for FPGAs. In *Proceedings of IEEE International Conference on Field-Programmable Technology*, FPT'03, pages 324–327, 2003.