Accelerating Hash Aggregate for Big Data Analytics

Anirban Nag
Huawei Zurich Research Center
Email: anirban.nag@huawei.com

***Abstract*—Big Data Analytics is a crucial workload for hyperscale service providers. On profiling analytics workload in modern ARM server systems, we see that 23% of the execution time is spent on hash aggregate operation. Thus, accelerating this operation can improve overall throughput of the system in processing analytical queries. In this paper, we first show that hash aggregate is a memory bound workload with data dependent memory accesses that stall the core and do not generate enough memory level parallelism (MLP). We introduce an accelerator with two novel techniques: (i) dataflow execution with pipeline of stages decoupled by hardware queues that allow producers (of memory accesses) to run ahead of consumers, (ii) a hierarchical locking mechanism that allow concurrent accesses to shared data structure (hash table) at high throughput. These innovations enable high memory bandwidth utilization which in turn leads to on average 52× speedup over a typical ARM server core.**

I. Introduction

With the end of Dennard scaling it is possible to shrink a transistor's size but not its power consumption. Accelerators have been widely adopted to circumvent this constraint. Die area that would be otherwise underused is instead invested in tailored, resource-efficient implementations of critical applications. Such accelerators have been successfully deployed for a variety of workloads including machine learning training and inference, web search, and network processing.

Data analytics is a key workload in server systems for performing business analytics. Analytics provide business insights to companies which helps them in making future decisions. Analytics services is a business sector with multi-billion USD market size [1]. Any improvement in throughput of analytical query processing can allow hyperscalers to process more volume (scale) with less resources. Companies like Intel and Oracle have proposed accelerators for data analytics (Intel In-Memory Analytics Accelerator [2] (IAA) and Oracle Data Analytics Accelerator [3] (DAX)) but they only support primitive operations that are less useful in production environment. In this paper, we highlight the importance of accelerating hash aggregate using custom logic to improve the throughput of data analytics.

II. Background

A. Big Data Analytics

Businesses typically store their data in Online Transaction Processing (OLTP) databases, which is in row format (all fields of a particular transaction are stored consecutively). This data is then transformed into columnar format (each field's data items are store consecutively) to enable faster analytical algorithms (and vector processing). A database consists of many tables, each comprising of multiple fields. An analytical query comprises of many different operations performed on the data, such as scan, filter, hash join, hash aggregate, sort, shuffle, etc.

The pie chart in Figure 1 shows the percentage of execution time spent on different operators (using Spark as the analytics software stack, running TPC-DS benchmark queries at scale factor 3000). The pie chart shows that a significant portion of the execution time (23%) is spent on hash aggregate operation (in green).

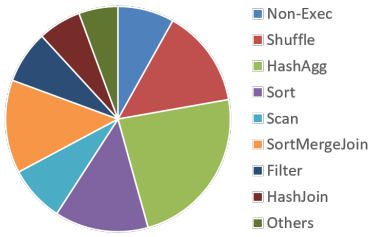

Fig. 1. Breakdown of hot functions in big data analytics.

B. Hash Aggregation

Aggregation (a.k.a group-by) involves scanning each row of a table and calculating aggregation statistics (sum, avg, min, max, etc) from the value column for each unique group in the group column. Such a task may involve one (or more) column(s) as input group, and one (or more) column(s) for aggregation value. To compute the aggregation, a hash based approach is preferred (over sorting the entire table) because it involves scanning the table only once. For each row in the table, the group column is hashed to find its corresponding hash table index. If there is an existing entry in the hash bucket that matches with the input group, the values are aggregated, else a new entry is added to the hash bucket. Hash collisions (two groups mapping to the same hash index) are resolved by chaining entries within a hash bucket, with allocations for new entries being obtained from an "Extended Table".

Figure 2 shows the hash aggregation process for an example SQL query. The query finds the aggregate number of units sold per unique salesman from the Sales table. When processing the last row (Salesman: Tom, Units: 1), the group column (Tom) is first hashed, which points to hash index 1. In that hash bucket, none of the existing entries match with group name "Tom", which is why a new entry is created (allocating an entry from the Extended Table) and chained to the base entry using a pointer.

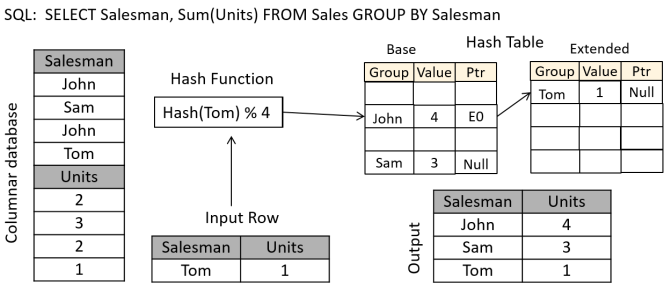

Fig. 2. Hash aggregation example query and execution steps.

C. Accelerator Ecosystem

Accelerators in server systems are generally integrated in the IO die. This is because the IOMMU unit [4] offer virtual memory functions (such as Address Translation Service and Page Request Service), which allow accelerators to work on virtual memory space without interrupting the calling core. Typically, an accelerator is invoked asynchronously (using for example co-routines) such that the core is not idle and can perform independent work. This requires software modifications, which can be easily obfuscated from the end user using analytics software engines (programmed by experts) provided by big tech companies (Velox [5] by Meta, OmniRuntime [6] by Huawei). Invocation is done using library APIs provided by the CPU vendor. Device drivers mmap hardware job queues such that a core can submit jobs. In ARM ecosystem, cores can atomically write to the job queue using ST64BV instruction and get a response back about success or failure in submitting the job. A job submitted to the accelerator sets up the different finite state machines (FSMs) by configuring the control registers. A DMA engine within the accelerator can directly access the memory for input, output, and intermediate data. This ecosystem across multiple stacks is depicted in Figure 3.

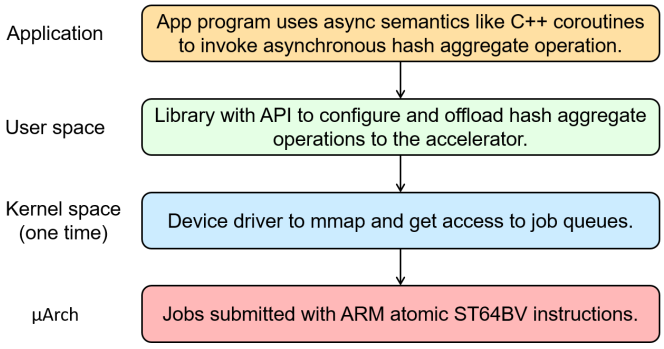

Fig. 3. Hardware/software interface to integrate accelerator in the ARM ecosystem.

III. The Hash Aggregate Accelerator

A. Workload Characterization

To guide us with an accelerator architecture, we construct a benchmark by isolating different hash aggregate kernels from the TPC-DS benchmark (with scale factor 500) with varying characteristics (different output groups cardinality, different input data types, number of aggregates per query, etc). In order to illustrate the key bottlenecks, consider a naive (first pass) accelerator with multiple hardware threads to increase the throughput. Each thread is responsible for serving one input row at a time, fetching hash table entry from the memory if required, performing the aggregation, storing the result in the cache, and evicting (and writing back) hash table entries from the cache if necessary. Each hardware thread is equipped with its own operand registers and ALUs.

Figure 4 shows the breakdown of execution time (obtained from a cycle-accurate simulator) for different hash aggregate kernels (sorted by their throughput). We see that most of the time the threads are waiting on memory accesses (reads: yellow, writes: blue) or idle (orange) because of load imbalance. This shows that hash aggregate is a memory-bound workload due to data dependent memory accesses to the hash table, whose performance is proportional to utilization of the available memory bandwidth. To achieve high throughput, such an architecture would need large number of threads to increase memory level parallelism (MLP). This model is not silicon (or power) efficient because precious silicon resources (registers) are wasted waiting on memory accesses. The architecture also needs large MUXes to share cache read/write ports across large number of threads.

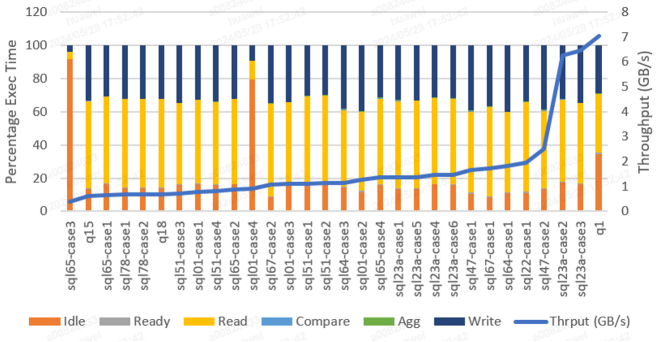

Fig. 4. Breakdown of time spent by different stages in a suite of hash aggregation kernels from TPC-DS.

B. Key Ideas

In order to design a silicon (and power) efficient architecture, we propose a novel dataflow architecture. In this model, application tasks are structured as a pipeline of stages decoupled by queues. Hardware queues hide latency effectively and allow scaling MLP for a memory-bound workload by allowing producer stages to run far ahead of consumers [7]. Instead of waiting on memory accesses occupying bulky registers, execution contexts (for processing input rows) wait on memory accesses being queued in area efficient hard queues (implemented in SRAM). Queues allow scaling MLP with fewer parallel execution context, which in turn enables silicon efficient implementation (smaller MUXes, fewer operand registers).

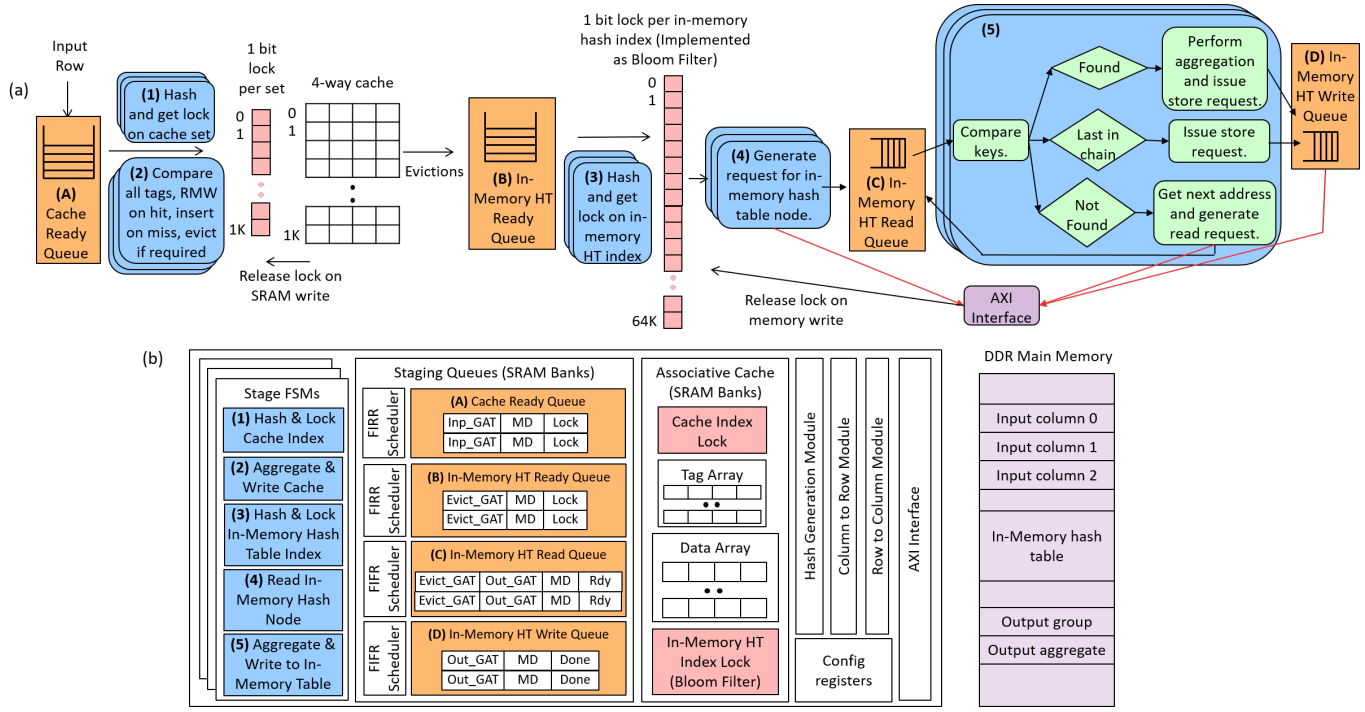

Fig. 5. (a) Hash aggregation execution flow depicted as a pipeline of stages (in blue) decoupled by hardware queues (in orange). Execution also uses hierarchical locks (in pink) to allow concurrent accesses to the hash table. (b) Architecture block diagram of the hash aggregate accelerator .

Figure 5(a) depicts the dataflow execution of hash aggregation. The execution comprises of multiple stages (blue boxes) set-up as a pipeline. Each stage is implemented as finite state machines (FSMs). The stages are decoupled using hardware queues (orange boxes). Each queue has a corresponding scheduler (First In First Ready (FIFR) or First In Round Robin (FIRR) that schedules items from the queue if available. All the stages run in parallel, getting an item from its input queue if possible, processing the item, and then either sending the item (or another item as part of cache evictions) to the output queue (if operation was successful) or keeping the item in its input queue to try again later. There are multiple copies of the pipeline (stacked blue boxes) to improve parallelism. In order to improve temporal locality of frequently encountered groups from the hash table, a set-associative cache is used with LRU/LFU eviction policy.

To enable concurrent accesses to the hash table without any correctness issues, we introduce a novel hierarchical locking mechanism (pink boxes). This includes a first level lock for cache accesses (1 bit lock per set) and a second level lock for each in-memory hash table index. Since the in-memory hash table is large, 1-bit lock per hash index can per prohibitively large in terms of silicon area. To reduce area overhead, a bloom filter lock is be used. Bloom filter is an approximate data structure that can have false negatives. The bloom filter may incorrectly classify an index as locked, which may not be performance optimal but it does not affect correctness. The bloom filter is designed as an array of counters, where insertion of an item (to get lock) leads to incrementing a set of counters, and evicting the same item (to release lock) leads to decrementing the same set of counters.

C. Architecture

Figure 5(b) shows the architecture block diagram of the accelerator. The pipeline stage FSMs and hardware queues are colored and numbered the same way as in Figure 5(a) (blue boxes (1-5) and orange boxes (A-D) respectively). Next we walk through the hash aggregation workflow using Figure 5 as reference.

A "Column to Row" module is used to fetch input data stored in columnar format and convert it to row format to be processed by the accelerator. The input row is then pushed to the "Cache Ready Queue" (Queue-A). An FIRR scheduler is used (for simplicity and fairness) to schedule a request to FSM-1. The "Hash & Lock Cache Index" stage (FSM-1) hashes the group column of an input row to determine the cache set and then tries to obtain a lock for the set. This is required to avoid race condition of multiple FSM-2 modifying the same cache-line. If a lock is obtained, the input row is forwarded to FSM-2, else it is enqueued back in Queue-A. The "Aggregate & Write Cache" stage (FSM-2) compares cache tags to find the matching entry from the hash table and does one of the following: (i) aggregate and writeback on hit, (ii) insert on miss, and (iii) evict an entry if no invalid entry is found in the set. Upon writeback, the lock bit is reset. The evicted entry is pushed to the "In-Memory HT Ready Queue" (Queue-B), which uses an FIRR scheduling to feed FSM-3.

Queue-B stores entries evicted from the cache (waiting to get a lock) that needs to be merged with entries in the in-

memory hash table. The "Hash & Lock In-Memory Hash Table Index" stage (FSM-3) hashes the group column of an evicted entry to find the corresponding hash table index and then tries to obtain a lock for that index using the bloom filter. Again, locking is essential to avoid race condition of multiple FSM-5 modifying the same hash index. If lock was successfully obtained, the entry is forwarded to FSM-4, else it is enqueued back in Queue-B. The "Read In-Memory Hash Node" stage (FSM-4) generates a read request (from memory) to read the base entry of a hash table index. The request is enqueued in the "In-Memory HT Read Queue" (Queue-C). Whenever the corresponding memory response is back, the request is ready to be scheduled (using FIFR scheduler) to FSM-5 for further processing. The "Aggregate & Write to In-Memory Table" stage (FSM-5) compares the group column of an evicted entry with that of the hash table entry and does the following: (i) if they are the same, aggregate and push a writeback request to Queue-D, (ii) if they are different, generate read request for the next hash table entry using the pointer chain and enqueue the request in Queue-C, (iii) if no match was found and the hash table entry is last in the chain, insert a new entry in the chain by pushing a writeback request to Queue-D. The "In-Memory HT Write Queue" (Queue-D) stores write requests to the hash table and marks them as complete whenever a write response is back (using FIFR scheduler).

When all the input rows have been processed and the pipeline is empty (all queues are empty), the in-memory hash table is read from memory and converted to columnar format using to the "Row to Column" module. The number of processing threads, the number of entries in each queue, the number of cache sets, and the number of entries in the in-memory hash table are configured based on the size of an input row (power-of-2 multiple of 32B input rows).

IV. Results

We developed an in-house cycle accurate simulator for the accelerator, taking into account DDR4 memory latency and bandwidth and also modeling the AXI interface. For our baseline, we use a typical ARM based server system, run multi-threaded parallel implementation of hash aggregate, and report normalized speedup over a single ARM core. Figure 6 plots the performance improvement using previously introduced hash aggregate micro-kernels from TPC-DS benchmark suite. The accelerator provides 26× geomean speedup (52× iso-area speedup), and maximum up to 135× speedup. The speedup is more pronounced for kernels with high output group cardinality. This is because for these cases, the hash table is large enough to not fit in the cache, resulting in many memory accesses that stall the CPU core. On the other hand, our accelerator is well suited in generating high MLP and maximizing the memory bandwidth utilization for cache unfriendly aggregation kernels.

V. Conclusion

In this paper, we first highlight the importance of accelerating hash aggregate to improve the performance of big data analytics, a key workload across hyperscale services in data-centers. We show that hash aggregate is primarily memory bound and performance is dependent on efficient memory bandwidth utilization. Thus, we introduce a dataflow architecture with pipeline of stages decoupled by hardware queues. Hardware queues provide a silicon efficient mechanism to allow producers of memory accesses to run ahead of consumers, thereby improving memory level parallelism. We also introduce a novel hierarhical locking mechanism to allow concurrent accesses to shared data-structure (hash table) at high throughput. The accelerator is 52× more performant than a typical ARM server core (iso-area). We also showcase the integration of such an accelerator in the ARM ecosystem across multiple layers of the stack. Future work includes adding support for more analytics operators (hash join, scan, filter, sort, etc) and supporting offloading a pipeline of operators.

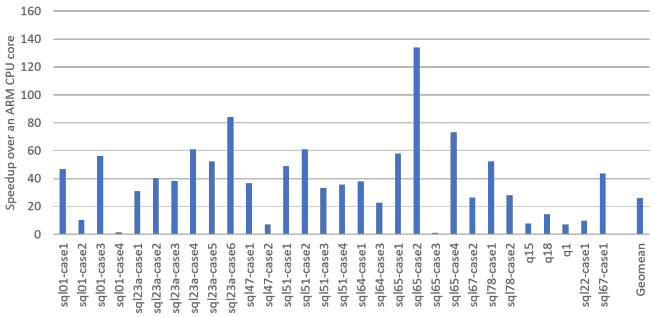

Fig. 6. Speedup in throughput of the accelerator over a typical server-based ARM core for a suite of hash aggregate kernels.

References

[1] S. Insider, "Big data analytics market to reach usd 842.6 billion by 2032 amidst rising demand for advanced data solutions." [Online]. Available: https://finance.yahoo.com/news/big-data-analytics-market-reach-130000382.html

[2] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeepan, J. Cabrera, A. Kwatra, R. Sankaran *et al.*, "Intel accelerators ecosystem: An soc-oriented perspective: Industry product," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 848–862.

[3] K. Aingaran, S. Jairath, and D. Lutz, "Software in silicon in the oracle sparc m7 processor," in *2016 IEEE Hot Chips 28 Symposium (HCS)*. IEEE, 2016, pp. 1–31.

[4] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "Iommu: Strategies for mitigating the iotlb bottleneck," in *International Symposium on Computer Architecture*. Springer, 2010, pp. 256–274.

[5] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay, "Velox: meta's unified execution engine," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3372–3384, 2022.

[6] Huawei, "Omniruntime overview." [Online]. Available: https://www.hikunpeng.com/document/detail/en/kunpengboostkithistory/240RC2/bds/kunpengbds_omniruntime_20_0002.html

[7] J. E. Smith, "Decoupled access/execute computer architectures," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, 1982.