# One Size Does Not Fit All: Accelerating OLAP Workloads with GPUs

Yansong Zhang[1,2] Yu Zhang[3] ✉ Jiaheng Lu[4] Shan Wang[1,2]
[1]DEKE Lab at Renmin University of China [2]School of Information in Renmin University of China
[3] National Satellite Meteorological Center of China [4] Department of Computer Science, University of Helsinki, Finland
zhangys_ruc@hotmail.com, yuzhang@cma.gov.cn, jiahenglu@gmail.com, swang@ruc.edu.cn

## ABSTRACT

GPU has been considered as one of the next-generation platforms for real-time query processing databases. In this paper we empirically demonstrate that the representative GPU databases (e.g., MapD[1]) may be much slower than another representative in-memory databases (e.g., Hyper[2]) with typical OLAP workloads (with Star Schema Benchmark) even if the actual dataset size of each query can completely fit in GPU memory. Therefore, we argue that GPU database designs should not be ONE-SIZE-FITS-ALL; a general-purpose GPU database engine may not be well-suited for OLAP workloads without tailored GPU memory assignment and GPU computing locality. In order to customize GPU OLAP, we need to re-organize OLAP operators and re-optimize OLAP model.

In particular, we first propose the *vector grouping* operator to achieve the maximal performance for AGGREGATION processing on top of query plan tree, and then we use *vector referencing* oriented star-join operator to serve for the upper *vector grouping* operation. The GROUPING operation is pushed down to the bottom dimension table scan nodes as group mapping operation to enable the optimal star-join and *vector grouping* operation in upper level of query processing tree. The inverted TOP-DOWN query plan tree optimization guarantees the optimal operation in final step and pushes the respective optimizations to the lower layers to make global optimization gains. Our experimental results show that the *vector grouping* operation achieves 2.7-15.55 times speedup over MapD CPU version and 1.39-5.81 times speedup over MapD GPU version with various grouping cardinalities. Further, the overall OLAP performance of proposed approaches is 2.23 times faster than MapD CPU version and 3.58 times faster than MapD GPU version for the Star Schema Benchmark (SSB) with scale factor 100.

## 1. INTRODUCTION

Nowadays in-memory databases are extensively adopted for high performance query processing as RAM sizes grow and prices are dropping dramatically. While the requirements of significant performance improvements are dominated by the low increasing of integrated core numbers, GPU databases are considered as another trend of high performance query processing engines with large amount of CUDA cores, high bandwidth device memory and scalability, e.g., the HGX-2[3] server can support 16 NVIDIA Tesla V100 GPU with total 0.5TB device memory and 300GB/s NVLink switch. The rapid developments of GPU push the developments of GPU databases. MapD, Kinetica[4], SQREAM[5], Zilliz[6], PG-Strom[7] are the representative GPU databases.

With the rapid developments of GPU databases, the following questions naturally arise.

● **First, can GPU databases beat in-memory databases with necessary optimization techniques and big enough memory size?**

In-memory databases, e.g., MonetDB, Actian Vector, Hyper are carefully optimized for memory hierarchy, cache and register. The column-at-a-time, vector-at-a-time and JIT compliant optimizations have been widely adopted as high performance database characteristics. However GPU databases have no revolutionary processing model or optimization techniques. Moreover, the limited GPU memory size and PCIe bandwidth also add additional overhead over in-memory databases.

● **Second, is the relational query processing model well-suited for GPU databases?** The Volcano iterative processing model is designed for pipeline processing on CPU platform assuming that the processing thread has enough private memory to cache intermediator materializations. CPU is designed with less cores but large L1-L2-L3 cache hierarchy to achieve high cache locality, while GPU is designed with massive cores but very small shared cache (shared memory). In other words, the GPU hardware can hardly match the traditional query processing model. Actually, GPU prefers to the small input/output and dense computing workloads instead of traditional query processing model of distributing sparse computing among a long query plan with large data stream.

● **Third, is the hybrid CPU-GPU database architecture or layered database architecture adaptive to GPU database design?** The core idea of GPU databases is to offload computationally intensive operations to the GPU cores by keeping the remainder of the operations running on the CPU cores [4]. State-of-the-art GPU databases commonly assign workloads among the equal CPU and GPU according to cost model as a fine-grained operation distribution optimization which may produce additional materialization overhead between CPU and GPU. As GPU databases are commonly used for data warehousing workloads [5], relational processing model is not the unique choice for GPU analytical processing. A different roadmap is to develop a layered database framework for CPU and GPU platforms with hybrid processing models. Ideally, we may divide the tight coupled relational processing model into different layers, organizing and offloading computing intensive workloads to GPU computing layer, remaining management intensive and data intensive workloads in different types of database layers. By layered framework, a coarse-grained strategy can be adopted for distributing workloads according to different characteristics of CPU and GPU platforms. Moreover, the GPU computing layer can be independently designed and implemented as GPU acceleration engine with loose

coupled relation to database engine, so that GPU acceleration engine can be considered as plug-in acceleration engine for databases and it can also employ different processing model or optimization techniques for GPU computing.

Therefore, although the GPU databases come to be industrial implementations, we still need a comprehensive evaluation for GPU databases on how well GPU databases perform for analytical workloads and exploiting how to design GPU acceleration engine to develop a layered databases.

In this paper, we focus on evaluating OLAP performance with the leading GPU database MapD and in-memory database Hyper, from which we can comprehensively evaluate the dominated relational operation performance and the whole query processing performance for MapD and Hyper. State-of-the-art researches focused on join algorithm optimizations [8-11] from multicore CPU platform to the emerging MIC Phi[12,13], GPU[14-16] and FPGA[17,18] platforms. The main-stream opinion is exploiting specified hardware features to improve join performance. At the same time, the industrial GPU databases show significant performance improvements against traditional databases with CPU, and MapD also contributes the open-source system to provide researchers a GPU database testbed. MapD supports both CPU mode as in-memory database and GPU mode as GPU database. The vectorizing query execution and JIT (Just-In-Time) compilation framework built on LLVM[19] combines the advantages of two leading in-memory databases Actian Vector[20] and Hyper, the GPU mode prefers to maximize GPU memory locality for hot data with a GPU memory resident style query processing. These optimization strategies enable MapD to be a high-performance in-memory database as well as GPU database. It is an intriguing topic to discover how MapD performs compared with the leading in-memory database Hyper. With a typical OLAP workload and GPU memory suitable dataset size, the expected result is that MapD wins, but the real situation is that MapD does lose the game.

In relational operation performance experiments, we find that both MapD CPU mode and GPU mode outperform Hyper with join and star-join operations, and MapD GPU mode also outperforms Hyper in grouping & aggregation operation. In another word, for two major operations in OLAP workloads, MapD beats Hyper, while Hyper outperforms MapD in the whole query processing performance. Paper [11] discovered that the join time may only be a 10%–15% share of the total runtime of a TPC-H query, the volcano query processing model and enhanced vectorized query processing model mix the join overhead among the whole query processing, so that improvements of join performance doesn't improve the whole query processing performance dramatically. We can partially conclude that the GPU database is well-suited for OLAP operation rather than OLAP query processing.

Due to the small GPU device memory size and low PCIe channel bandwidth, improving the GPU memory data locality is the key optimization during query processing. Paper [21] announced that on average only about 5% of execution time is spent on the GPU, we find that MapD only outperforms Hyper with simple queries. However, MapD is much lower than Hyper for complex queries with similar dataset size (under the GPU memory size). We develop a layered GPU database framework to combine multidimensional computing and relational processing by offloading the computing intensive star-join operation of OLAP to GPU acceleration engine. The layered GPU database framework with GPU acceleration implementation maximizes the data locality for both CPU and GPU. We also find that a simple GPU side star-join acceleration with CPU side aggregation proves to be more efficient than MapD.

The main contributions of this paper are summarized as follows:

1) We argue that GPU database performance is dominated by not only GPU memory size but also GPU memory utilization efficiency. The commonly adopted operator-at-a-time model in GPU databases suffers from high intermediate materialization overhead. Improving GPU memory utilization efficiency is as important as to improve performance.

2) We present the GPU OLAP acceleration model to customize GPU OLAP workload processing. The layered GPU database framework follows the divide-and-conquer rule to divide the whole query processing into three workloads with different processing features. By efficient vector grouping and star-join operations, the computing-intensive workload can be GPU memory resident and the data-intensive workload can be optimally assigned to CPU to minimize PCIe transfer overhead.

3) We design the top-down optimization instead of bottom-up query tree optimization. The normalized vector grouping operation first defines the optimal upper operation, then the lower operations are customized to support the upper vector grouping operation. By pushing early materialized grouping operation to the bottom, we can use vector index as shared intermediate materialization to improve GPU memory utilization efficiency in GPU side and improve grouping and aggregation operation in CPU side.

4) We design the star-join experiments with SSB workload to evaluate the join performance of leading in-memory and GPU database. Compared with MapD, our experimental results show that the vector grouping operation achieves 2.7-15.55 times speedup over MapD CPU version and 1.39-5.81 times speedup over MapD GPU version with various grouping cardinalities. In addition, for the Star Schema Benchmark (SSB) with scale factor 100, the whole OLAP performance is 2.23 times faster than MapD CPU version and 3.58 times faster than MapD GPU version.

**Organization** In Section 2, we briefly summarize in-memory databases and GPU databases. In Section 3, we discuss the layered database framework for hybrid CPU-GPU platforms. In Section 4, we describe the design of GPU star-join acceleration and vector grouping operation, which is used as GPU acceleration design opposite to GPU database design. We present the experiments in section 5. Section 6 reviews the related work and Section 7 concludes this paper.

# 2. BACKGROUND

In this section, we briefly describe and analyze in-memory databases and GPU databases.

## 2.1 In-memory Databases

MonetDB[22] is the pioneer column-store in-memory analytical database. The operator(column)-at-a-time processing model is much efficient than traditional tuple-at-a-time processing model, and the operator-at-a-time processing model is widely adopted by GPU databases[23]. MonetDB/X100[24] adopts the vector-at-a-time processing model to reduce materialization overhead against

column-at-a-time execution model. The vectorized processing model is now widely adopted by column-store in-memory databases, e.g., Actian Vector and MapD CPU mode. Hyper compiles queries into machine code using the optimizing LLVM compiler to achieve high query processing efficiency, and the register level optimizations enables Hyper to be high performance in data-centric workloads.

For in-memory databases, in-memory join algorithms are the dominated relational operation. Many previous works have studied the designs and implementations of different join algorithms facing state-of-the-art CPU architectures. As shown in Figure 1(A), multicore CPUs are commonly designed with L1-L2-L3 cache hierarchy, and improving cache locality is the core optimization for in-memory join algorithms. In the previous work [8], we have verified that NPO (no-partitioning hash join) is faster than PRO (Radix partitioning hash join) when the shared hash table is smaller than LLC(last level cache) size, while bigger table should employ PRO to partition both tables into cache fit partitions to perform in cache hash joins. Each core of CPU has its private L1 and L2 cache, and L3 cache is shared for all cores and threads. The hardware level auto cache replacement policy simplifies the multiple-threading programming for in-memory algorithm designs.

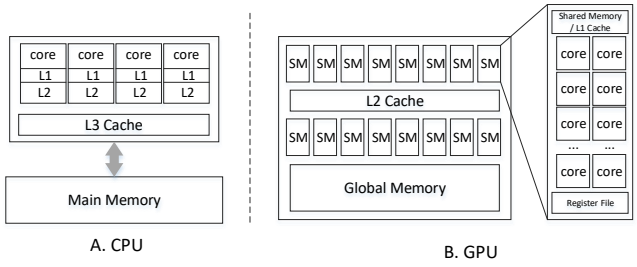



**Figure 1. Architectures of CPU and GPU.**

## 2.2 GPU Databases

As shown in Figure 1(B), GPU has different architecture from CPU. The on-board global memory is high bandwidth memory (HBM), the bandwidth is much higher than DRAM. GPU comprises with *streaming multiprocessor*(SM), and each SM comprises with many cuda cores. L1 cache/*shared memory* is shared for all cores inside SM. L2 cache is shared for all SMs. The L1 and L2 cache sizes are both much smaller than CPU's.

Programming on GPU uses a hierarchy of parallel threads, which are grouped into a wrap. Threads inside warp can efficiently synchronize with each other through shared memory. Programmers should manage to have each thread of a warp following the same control flow to improve GPU code efficiency. The global memory size of GPU is much smaller than DRAM size, e.g., the latest NVIDIA V100 GPU's memory size is 32 GB[3]. Memory transfer cost between DRAM and GPU through PCIe channel is the most critical bottleneck of GPU programming.

In a nutshell, GPU databases can be considered as extended in-memory databases, where the storage is DRAM and GPU's device memory, using both CPUs and GPUs as hybrid processors. The GPU end models are designed and optimized based on GPU architecture and cuda programming. The CPU end is used as host and schedules for GPU workloads. GPU databases need fundamental algorithm designs for GPU platform to make the maximal performance gains. The major in-memory relational operation algorithms should be re-designed for GPU platforms. Moreover, many mature cache-centric optimizations of in-memory databases are not adaptive to GPUs.

The general conclusion [23] of GPU researches is that GPUs prefer computing-intensive workloads with heavy computation on small dataset such as joins, and GPUs are not well-suited for data-intensive workloads with light computation on large dataset such as selections. The core idea is to hold computing-intensive data in GPU memory as much as possible. The limited GPU memory size also adds additional algorithm evaluation perspectives of memory efficiency, e.g., CPU platform prefers PRO join algorithms for higher performance at the cost of double memory consumption, while the similar join algorithm in GPU platform will decrease GPU memory utilization rate to support 50% less dataset.

MapD is the representative GPU database that can execute queries on either CPU or GPU platforms. It is designed with operator-at-a-time processing model with columnar storage. MapD tries to keep hot data in GPU memory as much as possible to improve GPU computing locality, and the typical configuration with multiple GPU cards (16 at most) can support 0.5 TB GPU memory for high performance GPU memory resident query processing. However, MapD applies a streaming mechanism for processing assuming that input data are not always fit in GPU' memory, the optimizer splits queries into chunks and assigns them to CPU or GPU according to the cost model.

As a summary, to achieve better query processing performance than in-memory databases, GPU databases should develop higher performance relational operations and minimize memory transfer cost.

# 3. Layered database framework

The hardware accelerators such as GPU, Phi, FPGA etc. come to be the first-class instruments for high performance computing architectures. The heterogeneous hardware devices are naturally divided into multiple layers, so that an ideal database architecture should also be divided into multiple layers to match hardware layers. In this section, we discuss how to design a layered database with OLAP domain knowledge.

## 3.1 Methodologies

Most GPU database systems focus on data warehouse workloads, we limit our research respective in OLAP database implementations.

For relational OLAP engine, the query processing is executed through query plan tree nodes as equal relational operations. From the multidimensional OLAP perspective, the tables and columns inside the queries are not equal for data locality and computing density. When we map ROLAP model to MOLAP model, the dimension tables are defined as dimensions and metadata for OLAP dataset, the foreign key columns are used as maps between dimensions and fact data to identify which fact tuple attends the following grouping & aggregation operation. As shown in Figure 2, the number of dimension tuples commonly increases slowly as data volume increases. The workloads on dimensions are management-intensive which includes updating dimensions and transforming SQL statements to multidimensional operations. The moderate workloads on dimensions are defined as warm workloads. Star-join is the performance dominated operation in OLAP, which is defined as Map operation in Figure 2 between dimensions and fact data. Star-join is in charge of mapping OLAP queries to fact data retrievals, which are commonly sparse computing on big data volume. Therefore, we define star-join workload as hot workload and define fact data retrieval workload as cool workload.

[25] discovered that in typical OLAP dataset the size of dimensions amounts to 1% the size of foreign key columns is about 19%, and the fact data size is more than 80% while the computing on foreign key occupies more than 80% of total execution time. Hence the OLAP workloads can be divided into 3 layers according to computing density. Moreover, the 3-layer workloads can match the typical database system architectures.

- For hybrid processor systems, the dimension management workloads can be assigned to CPU, the map computing can be assigned to CPU, GPU or FPGA to accelerate for dense star-join computing, the fact data retrieval workloads can also assigned to CPU, GPU or FPGA with optimal storage and computation efficiency.
- As GPUs are widely deployed as high performance cloud resources, we can also extend the 3-layer OLAP model in cloud computing platform. To achieve the optimal Total Cost of Ownership (TCO), GPU databases need not hold all the data in limited GPU memory, and the heterogeneous cloud resources can cooperate together. The dimension management workloads can be deployed in database cloud, and the map computing can be accelerated by GPU cloud and the fact data retrieval workloads can be deployed either on database cloud or data cloud for distributed and parallel computing.
- For database machine systems such as Oracle Exadata or IBM Netezza, the 3-layer OLAP model is also adaptive to the asymmetrical hardware architectures. The central database nodes are in charge of dimension management, the accelerator nodes do the performance dominated map computing and the scalable storage nodes are used for fact data retrieval workloads.

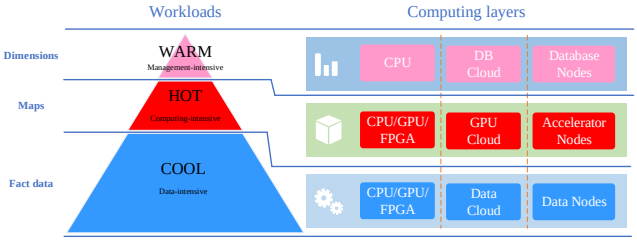



**Figure 2. Example for layered OLAP database.**

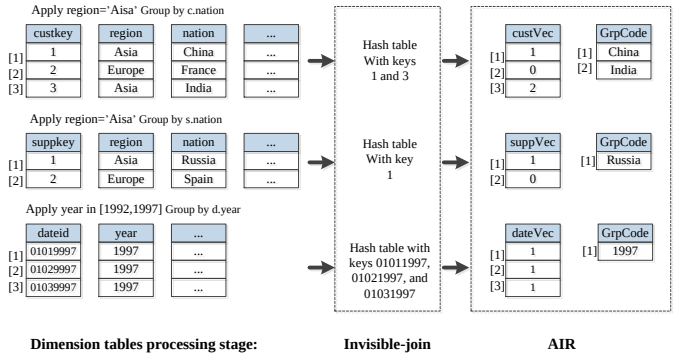
As a summary, the 3-layer OLAP model, which motivates the workloads distribution strategy from OLAP model perspective, shows that the computing density and data volume features are key considerations to assign layered workloads on heterogeneous computing platforms.

## 3.2 Star-join model

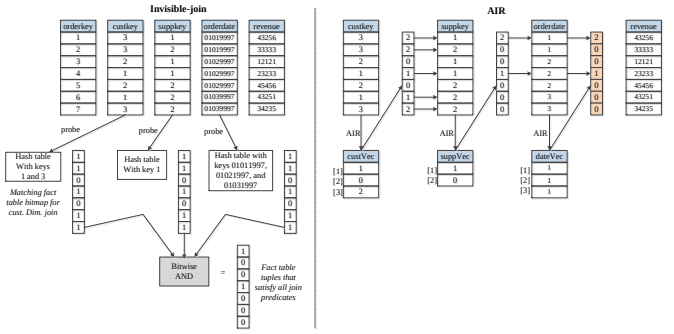
In 3-layer OLAP model, the hot workload is the mapping operation which is represented as star-join. However, traditional star-join is not well-suited for GPU acceleration for commonly adopted operation-at-a-time model which consumes too much intermediator materialization overhead for limited GPU memory. In the following section, we discuss two star-join models to illustrate how to perform a memory efficient star-join for GPU.

The invisible-join [32] model is a representative operation-at-a-time model for OLAP with late-materialization strategy, the OLAP processing can be divided into 3 stages. The invisible-join model first creates hash tables for dimension tables with keys to server for the following star-join operation. The star-join generates a bitmap to identify how to perform the fact data retrieval with computed positions. As the bitmap has no information of GROUP-BY attributes from dimension tables, the additional joins are executed to materialize join results for the following aggregation. For typical operation-at-a-time model, the materialization cost is expensive especially for limited GPU memory size.

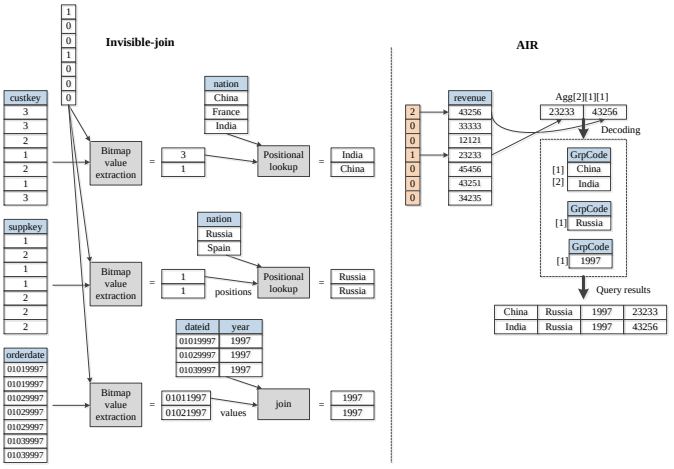
As shown in figure 3, AIR algorithm [25] presented a similar 3 stage processing model. The major differences laid out as follows: 1) in dimension computing stage, the dimension tables produce dimension vector instead of traditional hash tables; 2) in star-join stage, the AIR (array index referencing) algorithm performs the efficient star-join operation instead of hash based pipelining multiple table joins; 3) in fact data retrieval stage, the join result vector directly performs positional lookup on fact columns and aggregation cube based aggregation.




**A. dimension computing**




**B. star-join**




**C. fact data retrieval**

**Figure 3. Example for two OLAP implementations.**

AIR algorithm is designed with OLAP domain knowledge to fuse MOLAP model and ROLAP model together, how to maintain MOLAP features during updates and how to use the Fusion OLAP model is discussed in [26].

Without Fusion OLAP model constraints, the traditional SQL engine can also perform a similar layered processing model with trivial tricks. For a general purpose relational engine, the OLAP processing can be implemented with 3 alternatives with vector index grouping.

Figure 4 shows the hash based vector index grouping method. The traditional hash joins between fact table and dimension tables are performed, at the end of star-join stage, the GROUP-BY attributes are used to produce vector index instead of hash aggregation. The Group ID generator is employed to assign consecutive IDs for each group member with latch structure to guarantee assigning unique incremental ID for each new GROUP-BY hash tuple. A hash table is used to generate vector index, the GROUP-BY attributes in each join result is probed in hash table, if the GROUP-BY attributes are not matched in the hash table, the new hash tuple is created with a Group ID assigned from Group ID generator, at the same time the Group ID is written to the corresponding vector index cell and the GROUP-BY attributes are recorded in group vector cell mapping by Group ID; if the GROUP-BY attributes are matched in the hash table, just write the matched Group ID in the corresponding vector index cell. Finally, we get a vector index and a group vector as join results. The vector index can be further compressed with [FID, GID] tuples to eliminate null cells in vector index when selectivity is low. Now, we can perform a vector index based aggregation on fact table. We can perform a positional lookup on measure columns with address from non-NULL vector index cell or FID from compressed vector index, the measure column values are directly mapped to vector aggregator cell by vector index cell value or GID for aggregation. By merging group vector in star-join stage and vector aggregator in fact data retrieval stage, we get the final OLAP results.

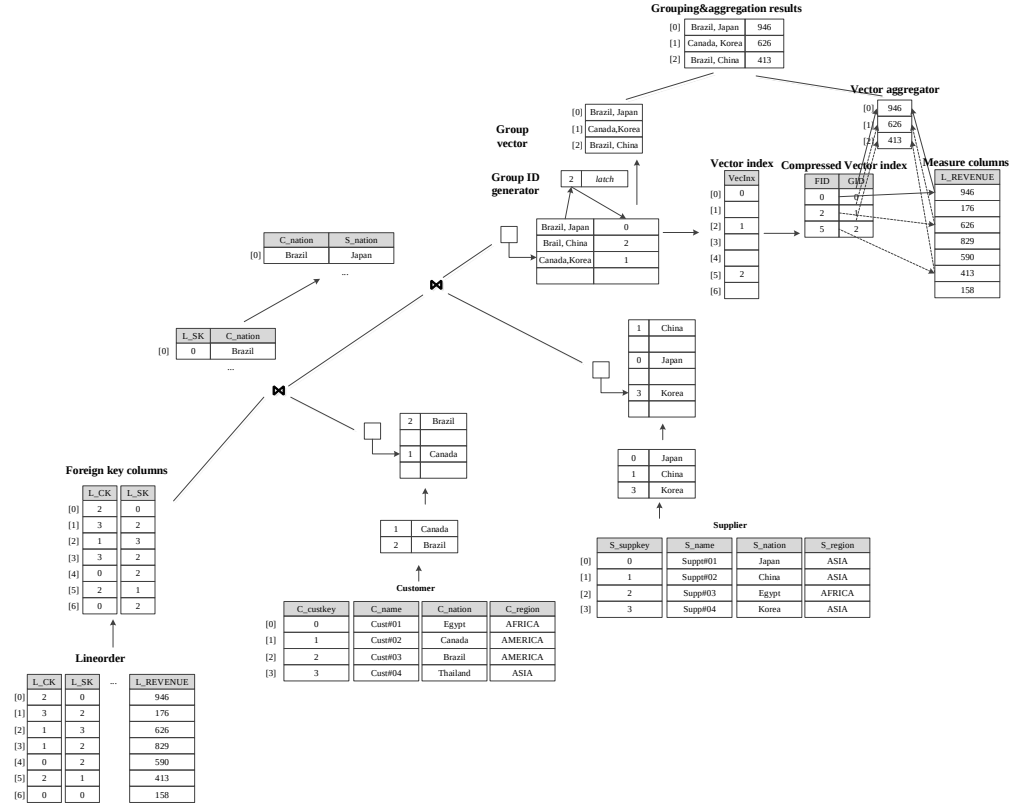



**Figure 4. Hash based vector index grouping.**

Hash based vector index grouping uses hash table to produce vector index, the low cardinality GROUP-BY attributes have to be repeatedly computed for hash keys.

Figure 5 shows a cube based vector index grouping method. The filtered and projected attributes in dimension table processing stage are compressed with dictionary compression, the dictionary vector index are used as Group ID for current dimension table. For multiple GROUP-BY attributes in single dimension table, each distinct GROUP-BY attributes pair is compressed as single Group ID. So that, the join results are Group IDs instead original long GROUP-BY attributes. Furthermore, the dictionary vectors of each dimension table construct a multiple dimensional array with each Group ID mapping to one sub-dimension array index. During star-join stage, the Group IDs are directly mapping to cell of group cube, the cube cell address is transformed as 1-dimension array index and the array index is written to the corresponding vector index cell. The 1-dimension array is used as group vector for aggregation with vector index or compressed vector index oriented positional lookup and aggregation on measure columns. When we get the final group vector, each non-null cell index is transformed into multidimensional address, and each dimension address is mapped to corresponding dimension dictionary vector to access the original GROUP-BY attributes, the combination produces the final OLAP results.

The cube based vector index grouping method is efficient because the multidimensional mapping takes the place of CPU

cycle consuming hash probing. General OLAP queries use small cardinality groups for interactive analytical processing, the group cube is commonly small and dense for cell utilization. If OLAP query produces a big cube while the cube is sparse in use, we can further optimize the group cube by mapping the non-null cells to a dense vector.

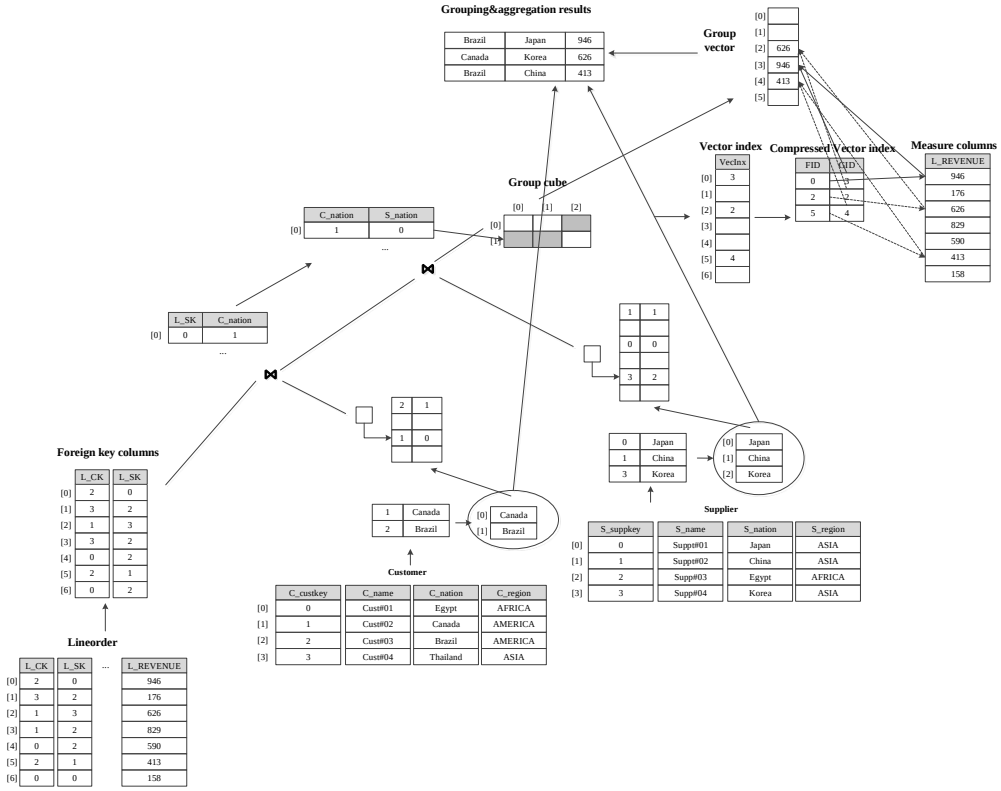



**Figure 5. Cube based vector index grouping.**

Figure 6 illustrates a vector based vector index grouping method. As the group cube is big and sparse, we employ a group vector to map final GROUP-BY attributes member. We also use Group ID generator to assign the unique ID as GID, so that we can get a dense vector instead of the sparse cube. As illustration, the group cube is mapped to a 1-dimension group vector, if the corresponding cell is null, getting an ID from the Group ID generator and writing the ID in the corresponding position of vector index, if the corresponding cell is already assigned an ID, just writing the ID into corresponding vector index cell. When we get the final vector index, we also get the maximal ID from Group ID generator, and we create a vector aggregator with the length equal to the maximal Group ID as aggregator for measure column computing. During the vector index or compressed vector index oriented scan on measure columns, the corresponding measure column attributes are mapped to vector aggregator cells for aggregating. Finally, the OLAP results are obtained by dual mapping with group vector. The non-null cells in group vector are mapped to dimension dictionary vectors with vector address to get the original GROUP-BY attributes, then the non-null values are mapped to vector aggregator cell to get the aggregation results, the combination of these two mapping results produces the final OLAP results.

The vector based vector index grouping method generates the minimal vector aggregator size during aggregation computing.

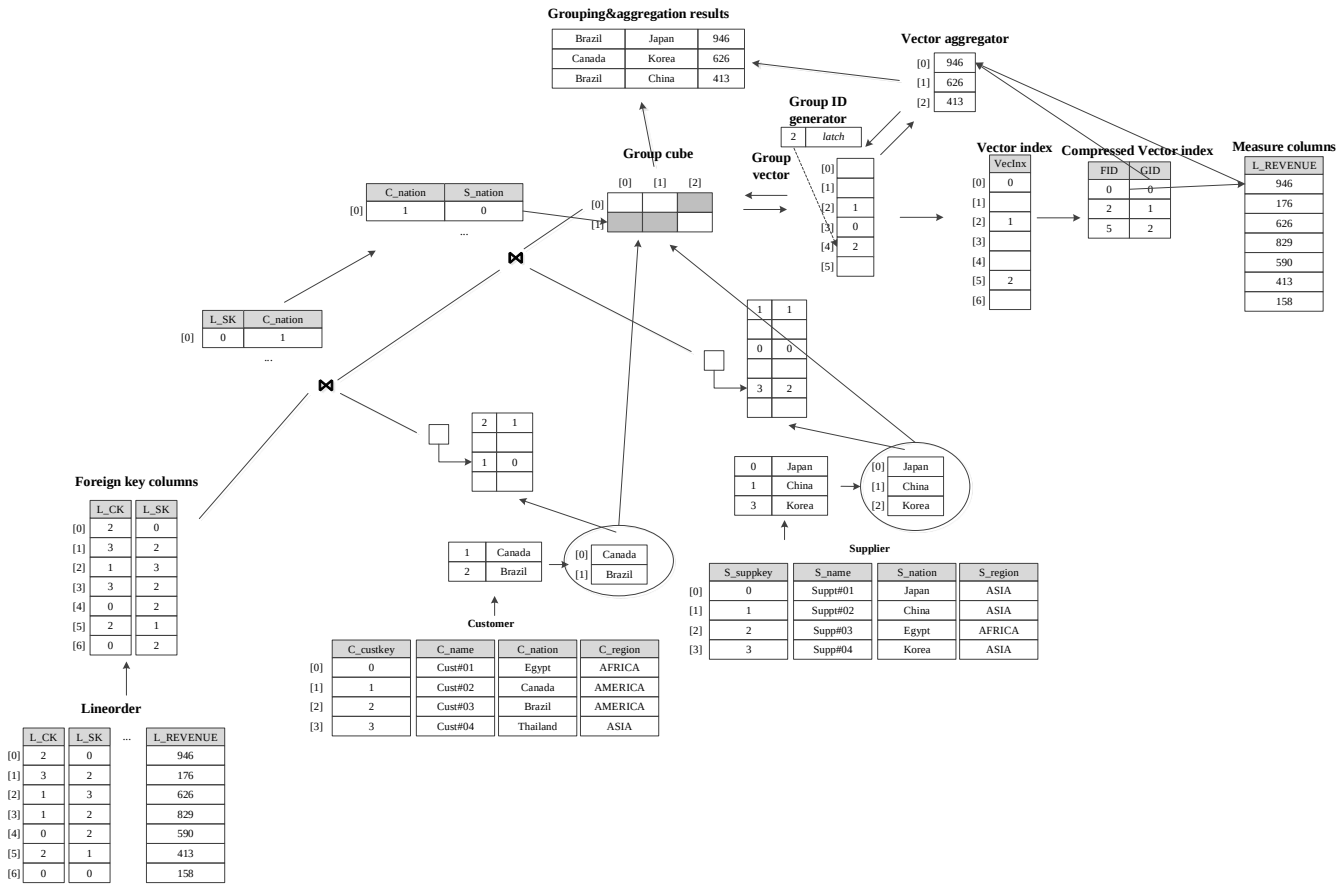



**Figure 6. Vector based vector index grouping.**

With the three vector index grouping methods, the traditional pipeline based query processing model can also be optimized as 3 stage processing on dimension processing, star-join and aggregation. The vector index is used as an intermediator between star-join and aggregation stages to divide the pipeline processing into independent processing stages. In another word, traditional SQL engines can also support 3-layer OLAP model like AIR by adding additional modules for dictionary compression, vector index, group vector, etc.

The three processing stages can be matched with 3-layer OLAP model, the different stage can be assigned to corresponding processing stage with different platforms. For example, the dimension table processing stage can be assigned to a full-fledged database engine as query scheduler, the star-join stage can be assigned to acceleration layer with GPUs with moderate dataset size and dense computation, and the fact data retrieval stage can be assigned to CPU or FPGA platforms with simple computation on large dataset. The layered OLAP model simplifies the database optimizer engine, the dimension table size, foreign key column size and fact data size are fixed or predicted for given OLAP workloads, the computation feature for the different datasets can be predicted, and the data transfer between each layer is fixed or predicted, the optimizer engine can give a simple optimization for hybrid CPU and GPU platform. The layered OLAP model also makes database flexible for heterogeneous platforms. The dimension table processing stage and fact data retrieval stage can be performed with database engine, and the star-join stage can be accelerated by CPU or high performance and scalable hardware accelerators like GPUs or FPGAs.

In next section, we focus on CPU-GPU hybrid platforms and compare GPU acceleration implementation based on 3-layer OLAP model and state-of-the-art GPU databases.

# 4. GPU Acceleration

GPU databases are commonly used for data warehousing and OLAP scenarios, besides GPU hardware-conscious optimizations for relational operation implementations, we can further accelerate GPU OLAP performance with a multidimensional perspective.

## 4.1 Multidimensional query plan tree

A typical OLAP query is to join fact table with filtered multiple dimension tables together and group the joined tuples for aggregation, e.g., Q4.1 in SSB:

```
Select
d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
From date, customer, supplier, part, lineorder
Where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation;
```

The relational query plan tree is shown in Figure 7(A), $\varepsilon_{d\_year,\ c\_nation}\Sigma$ denotes grouping&aggregation operation. During query processing, the upper operator iteratively "pull" tuples from the lower operator, the pipelined processing model eliminates the materialization overhead between operators. However, this pipeline query processing model involves much function call overhead and sacrifices the data locality of upper operator. One improvement is using vectorized processing model to process vectors instead of single tuple. The L1 cache fit vectors are processed as a batch to share the function call overhead, and the L1 cache level materialization overhead is trivial for the whole query processing. The vectorized processing model is adaptive to CPU architecture in which each core has its private L1, L2 cache. The other improvement is to employ JIT compliant technique to transform the "pull" mode to "push" mode. This effort uses Just-In-Time compiled low level machine code to improve code efficiency and the register level optimization improves the data locality.

GPU databases commonly adopt operator-at-a-time processing model, the materialization overhead is even critical than in-memory databases due to the limited GPU memory size. We find in the experiments that even if the original dataset size can be held in GPU memory, MapD doesn't prove high performance as GPU resident processing.

Based on the GPU's limited memory size, we revise the relational query plan tree as multidimensional query plan tree which is shown in Figure 3(B). First of all, multidimensional query plan tree follows an OLAP domain knowledge customized design to divide the whole OLAP query processing into 3 computing stages over specified dataset.

Stage 1 is assigned to dimension tables, the OLAP query is rewritten as mapping selection and grouping operations to dimension vector, the group-by clause is used to generate a grouping vector which can be considered as pushing the grouping operator in relational query plan tree into bottom node. This change eliminates iteratively pulling grouping attributes from the bottom nodes and the materialization overhead during column-wise processing, the vector grouping in the upper node also transforms hashing overhead on the massive grouping attributes as efficient vector grouping operation.

Stage 2 limits the star-join of OLAP within foreign key columns of fact table and involved dimension vectors. The OLAP schema dominates that star-join is the central and fundamental operation of OLAP, [25] discovered that the star-join operation shares average the 86% total runtime of SSB queries over about 20% total size of all 5 tables, and the following grouping&aggregation operation involves large fact columns with only 6.5% share of total runtime. The relational query plan tree pipes join and grouping&aggregation operations to GPU processor which will reduce GPU memory utilization rate for computing-intensive star-join workload and underutilize GPU's computing power. Furthermore, the join algorithm on GPU is simplified from *hash join* to *vector referencing*[26], this revision dramatically reduces the redundant hashing overhead for the many-to-1 mapping between foreign key and referenced primary key. In Figure 7(B), $\psi$ denotes mapping selection and grouping operations to dimension vectors; $\Omega$ denotes pre-generate vector for grouping; $\rightarrowtail$ denotes *vector referencing* operation; $\rightrightarrows$ denotes vector grouping operation.

Stage 3 binds the vector grouping operation to large fact columns which are accessed in queries. As the grouping&aggregation operation is simplified as vector grouping, it should be carefully evaluated which platform is better.

The 3-stage-computing model of multidimensional query plan tree groups the multiple relational operators into 3 OLAP operations: dimension maping, star-join and vector grouping, the GPU OLAP cost model is simplified as assigning the 3 stage to proper platform.

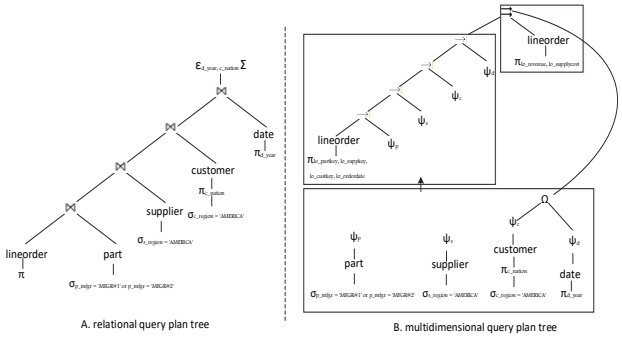



**Figure 7. Relational query plan tree vs. multidimensional query plan tree.**

## 4.2 Star-join acceleration

For star-join with operator-at-a-time processing model, the critical issue is to reduce materialization overhead. In this paper, we reuse vector index as the shared materialization for star-join, the shared vector index column is used as template vector index for all OLAP queries with different value distributions.

As shown in Figure 8, an example OLAP query invokes fact table joining with 3 dimension tables with 3 grouping attribute from the 3 dimension tables. In stage 1, the query is rewritten to generate 3 dimension vector named *DimVectorIndex*$_i$, the group-by attribute are projected and organized as a 3-*D* cube for aggregation, the indexes of the cube is encoded into corresponding dimension vector. In stage 2, the foreign key is mapped to dimension vector index, the traditional hash join is simplified as vector referencing operation, the *VectorIndex* is used to store join result for each foreign key item, the cell is either *NULL* or incremental multidimensional address of current tuple. So that, each pass of star-join shares and incrementally updates the *VectorIndex*.

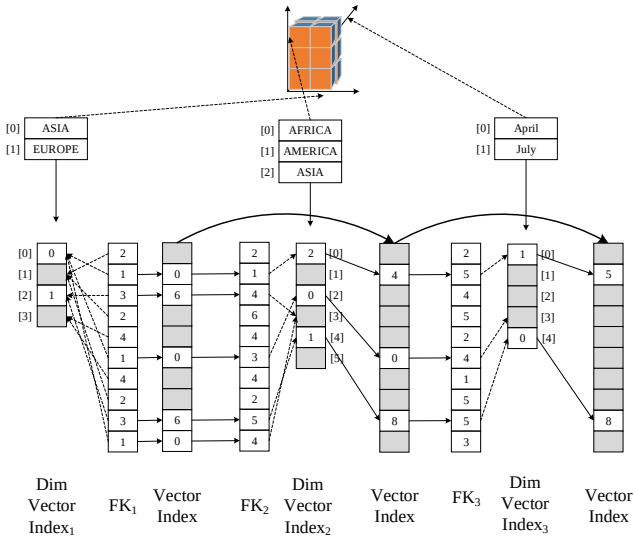


**Figure 8. Example for star-join implementation.**

For GPU star-join acceleration, we assign the available GPU memory to store foreign key columns and *VectorIndex*, the small size *DimVectorIndex* vectors are on-the-fly transferred to GPU memory. Let *GMS* denote GPU memory size, *FKWi* denote foreign key column width, *VIW* denote *VectorIndex* width, $DVW_j$ denote *DimVectorIndex* vector width, $DVL_j$ denote *DimVectorIndex* vector length, the rows *R* of GPU memory store can be calculated as:

$$R = \frac{GMS - \sum_j^m (DVW_j \times DVL_j)}{(\sum_{i=1}^n FKW_i + VIW)}$$

So, for given OLAP dataset, we can accurately know how many GPU cards we need. The GPU star-join acceleration design only accelerates computing workloads on 20% dataset size which dramatically relaxes the constraints for processing large dataset on GPUs.

The GPU programming for star-join is straightforward, the foreign key columns are parallel accessed by GPU threads, the *DimVectorIndex* vector is random accessed by parallel threads, and the results are written to corresponding *VectorIndex* cells by each thread without synchronization overhead. Current GPU star-join implementation uses fixed length as star-join template for queries with various selectivities. For low selectivity queries, the dynamically allocated *VectorIndex* may be more efficient due to we can only store (OID, VALUE) pairs to reduce scan cost on *VectorIndex*. The optimization adds GPU memory allocating overhead and synchronization overhead during dynamically assigning *VectorIndex* cells for parallel threads. Moreover, OLAP queries commonly involves a group of queries with selectivities from low to high or high to low as rollup or drill-down operations, the fixed length *VectorIndex* implementation is adaptive to maintain overall GPU memory utilization.

## 4.3 Vector grouping

OLAP queries commonly produce limited groups for interactive analysis, and the hash aggregation is typical implementation. For small groups, each thread can maintain private hash table for local aggregation, for large groups, threads can use shared hash table for global aggregation, which is well-suited for multicore CPU' architecture with L1-L2-L3 cache hierarchy and low synchronization overhead. On the contrary, GPUs share small shared memory for massive threads inside SM, and the synchronization overhead between threads is also high.

Most important of all, aggregation operation commonly involves large fact dataset with low selectivities, the memory transfer cost may be higher than GPU aggregation performance gains.

In this paper, we experimentally study the vector grouping performance for OLAP workloads of SSB to exploit how MapD, Hyper, vector grouping behaves for different workloads, and evaluate how to assign grouping workloads between CPU and GPU.

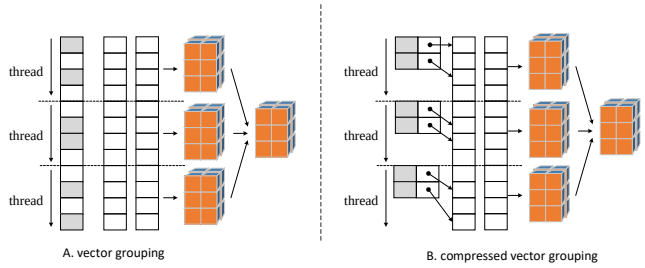



**Figure 9. Vector grouping.**

With the vector index, the fact columns can be logically partitioned into chunks and each chunk is assigned to one thread. The threads parallel scan the vector index, accessing the fact column cells according to *non-NULL* cells in vector index, and aggregating them in private aggregation cubes. Finally, the private aggregation cubes are merged together for global aggregation cube.

Shown as figure 9, when aggregation cube is larger than private cache size of CPU core, threads can concurrently update the shared aggregation cube for global aggregation with concurrent control mechanism.

For queries with very low selectivity, the vector index can be compressed with *non-NULL* OID and value pairs, the sequential scan on vector index is optimized as efficient positional scan.

Aggregation cube is customized data structure for OLAP by organizing group-by clause as multidimensional cube. For relational model, the mechanism can be implemented by pushing grouping operation down to dimension table scan node and compressing grouping attributes with dictionary compression by assigning each grouping attribute one consecutive number as dimension index. In star-join stage, the multidimensional cube address can be iteratively computed as one dimension vector for grouping and aggregation.

## 4.4 GPU OLAP acceleration model

The ideal scenario of GPU databases is GPU memory resident processing without data transfer cost for the maximal GPU computing efficiency. The general solution is to employ a cost model to decide whether an operation should be executed on GPU or CPU to achieve better performance. From the relational perspective, operators have various cost in different queries, it's hard to make a global processing model for different queries. From the multidimensional perspective, the OLAP dataset is a simple big table(fact table) with meta data(dimension tables) and multidimensional relations(foreign key columns in fact table). The eventual OLAP query plan is retrieving tuples from fact table and pushing them to cube for aggregation, which can be translated as performing aggregation with vector grouping operation in relational database. The star-join operation can be defined as mapping query parameters to vector index, and can be accelerated by GPU.

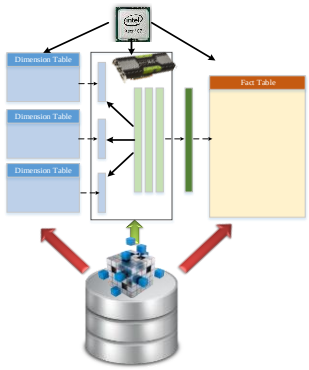



**Figure 10. GPU OLAP acceleration.**

Figure 10 illustrates the GPU OLAP acceleration model, in which the dataset is organized as combination of relational model and multidimensional model. The GPU star-join acceleration represents the multidimensional computing, and the processing on

dimension tables and fact table are performed by traditional relational database engine.

The GPU OLAP model can accurately define the data distribution between CPU and GPU memory, the computing workloads for CPU and GPU, the maximal utilization rate of GPU memory, and it can also guarantee the GPU memory resident dataset performing an in-GPU-memory computing without additional materialization overhead as operator-at-a-time processing engine.

# 5. EXPERIMENTAL EVALUATIONS

The purpose of the following empirical experiments focuses on three issues: (1) How well the GPU databases perform when compared with in-memory databases? (2) How well the GPU operations perform when compared with in-memory database operations? (3) Can GPU acceleration be superior to GPU database for OLAP workloads?

## 5.1 Experimental setting

The experiments were performed on a Supermicro Super Workstation 7047GR-TPRF with one Intel Xeon E5-2699 v4@2.2GHz CPU and 256 GB DDR4 RAM. The CPU has 22 cores and 44 physical threads. The OS is CentOS 7, and the Linux kernel version is 3.10.0-514.16.1.el7.x86_64. The GCC compiler version is 4.8.5. The server equips with a NVIDIA K80 GPU with two GK210 GPUs. K80 GPU comprises with 4992 CUDA cores and 24 GB GDDR5 on-board memory. The bandwidth of GPU memory is around 480 GB/s.

Hyper is downloaded from [27], the Actian Vector 5.0 is download from the action website (http://esd.actian.com/), and MonetDB is downloaded from MonetDB website (https://www.monetdb.org/downloads/). Hyper, Vector, MonetDB are used as representative in-memory databases. We use MapD[28] as representative GPU database from Github, and MapD is configured as GPU version, we can use "/GPU" or "/CPU" commands to switch to CPU mode or GPU mode. By "/memory_summary" command, we can see MapD uses two GPUs of K80 with total more than 20 GB GPU memory. PG-Strom is used as another representative GPU database downloaded from [29], we configured 4 modes as CPU-only, with GPU, CPU-only with in-memory mode, with GPU and in-memory mode. The *vector referencing* based join and star-join algorithms are employed from [25] and [26], we also develop the vector grouping algorithm in the open source code from [8] with the same *latch* mechanism for shared vector updating.

## 5.2 OLAP workloads

In our experiments, we used Star Schema Benchmark (SSB) as OLAP workloads. SSB is the denormalization design for TPC-H, the star schema is compatible with multidimensional model for OLAP analysis. The grouped queries vary from high selectivity to low selectivity to simulate drilldown or rollup operations, the group-by clauses simulate interactive OLAP queries with group amount various from 1 to 800. MapD doesn't support complex queries like subquery in TPC-H, the common query is with standard SPJGA operations like SSB.

We use SF=100 as experimental dataset, the numbers of rows of tables are 2555 (Date table), 200,000 (Supplier table), 1,400,000 (part), 3,000,000 (customer), 600,038,144 (lineorder), the detailed query parameters are shown in Table 1. We calculate the column sizes for each query, and we repeatedly execute the query 3-5 times in MapD to make columns that are accessed in query stay in GPU memory. The actual column sizes of 13 queries with dataset of SF=100 are no more than 20GB which can be held within MapD's GPU memory allocated from K80's 24GB memory.

**Table 1. Query parameters for SSB**

| Queries | Join tables | Selectivity | Groups | Size(MB) |
|---|---|---|---|---|
| **Q1.1** | LO⋈D | 0.019 | 1 | 9156 |
| **Q1.2** | LO⋈D | 0.00065 | 1 | 9156 |
| **Q1.3** | LO⋈D | 0.000075 | 1 | 9156 |
| **Q2.1** | LO⋈D⋈P⋈S | 0.008 | 280 | 9254 |
| **Q2.2** | LO⋈D⋈P⋈S | 0.0016 | 56 | 9211 |
| **Q2.3** | LO⋈D⋈P⋈S | 0.0002 | 7 | 9211 |
| **Q3.1** | LO⋈D⋈S⋈C | 0.034 | 150 | 9363 |
| **Q3.2** | LO⋈D⋈S⋈C | 0.0014 | 600 | 9363 |
| **Q3.3** | LO⋈D⋈S⋈C | 0.000055 | 24 | 9266 |
| **Q3.4** | LO⋈D⋈S⋈C | 0.00000076 | 4 | 9266 |
| **Q4.1** | LO⋈D⋈S⋈P ⋈C | 0.016 | 35 | 13892 |
| **Q4.2** | LO⋈D⋈S⋈P ⋈C | 0.0046 | 100 | 13989 |
| **Q4.3** | LO⋈D⋈S⋈P ⋈C | 0.000091 | 800 | 13941 |

LO: lineorder, D: date, S: supplier, P: part, C: customer

## 5.3 Benchmark performance of In-memory and GPU databases

We first evaluate the Benchmark performance of PG-Strom. As PostgreSQL is a disk-resident database engine, we test the original Benchmark performance of PostgreSQL, then test for PG-Strom

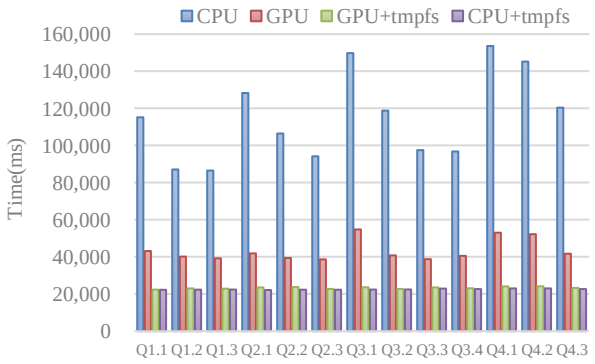



**Figure 11. PG-Strom Query Execution Time for SSB(SF=100).**

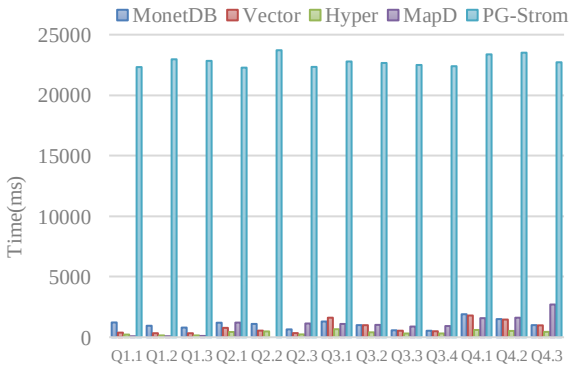



**Figure 12. Query Execution Time for SSB(SF=100).**

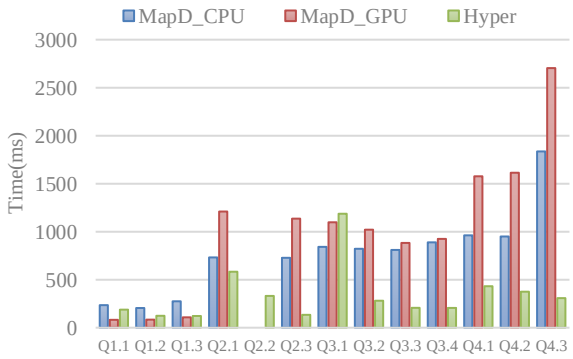



**Figure 13. Performance comparison of MapD(CPU/GPU) and Hyper with SSB(SF=100).**

with GPU. We find that PG-Strom can accelerate query processing performance, the average query execution time of PostgreSQL is 2.66 times of PG-Strom. Consequently, we use *tmpfs* to simulate in-memory PostgreSQL and PG-Strom performance by loading data in *tmpfs*. As shown in Figure 11, we find that the performance gains of PG-Strom is lost when compared with in-memory PostgreSQL mode, the average query execution time of PG-Strom is even a little longer than PostgreSQL. The results show that GPUs in PG-Strom offload computing workloads from PostgreSQL engine so that PG-Strom can process the data access and query processing workloads in parallel to reduce query execution time. However, when we use *tmpfs* to eliminate the I/O bottleneck of PostgreSQL, the PG-Strom still faces the PCIe bottleneck by transferring data from main-memory to GPU's device memory for processing. With sufficient memory, the in-memory PostgreSQL is superior to PG-Strom with GPUs. If the memory size is not large enough to hold all the data, PG-Strom with GPUs can achieve well performance improvements from disk-oriented PostgreSQL.

When we compared PG-Strom with typical in-memory databases and GPU database, we find that the performance of PG-Strom is dramatically lower than MonetDB, Actian Vector, Hyper and MapD(GPU mode) as shown in Figure 12. Although all the candidate databases are memory resident, the column-store, vectorized processing, JIT compilation and other specified optimizations of MonetDB, Actian Vector, Hyper and MapD beat the original row-wise PostgreSQL engine.

Among in-memory databases, Hyper achieves the highest performance in SSB experiments for the JIT compilation and register level optimizations. We use Hyper to represent state-of-the-art in-memory databases to detailed analyze the performance between Hyper and MapD in the following sections.

## 5.4 CPU and GPU mode of MapD

In SSB testing, Hyper occurs the numeric overflow error for *sum*() function, we alter the aggregate function with *count*(), the average execution time of MapD is about tens milliseconds(3% less for CPU mode and 10% less for GPU mode) less than original execution time.

For MapD GPU mode, the first time to run query is very slow with memory transfer cost through PCIe channel. We repeatedly run the test query 3-5 times to get the shortest time as MapD GPU mode execution time. Within the 13 queries, MapD fails to run Q2.2 due to "Cast from dictionary-encoded string to none-encoded would be slow" error both for CPU and GPU modes. So we neglect the Q2.2 execution time.

In Figure 13, the average execution time of MapD GPU mode is 1037 *ms*, MapD CPU mode is 774 *ms*, and the average execution time of Hyper is 346 *ms*. Hyper outperforms MapD both for CPU mode and GPU mode. Additionally, the average execution time of MonetDB and Actian Vector are 1052 *ms* and 837 *ms*. MapD adopts vectorized processing technique like Actian Vector, and the performance is also similar, and MapD outperforms MonetDB with operator-at-a-time model.

K80 GPU has 24 GB device memory, and MapD allocates more than 20GB, the size of query involving columns for each query is also smaller than 20GB, the repeatedly executed query enables MapD keep dataset in GPU memory, but MapD GPU mode is still slower than Hyper and MapD CPU mode.

We analyze the query execution time and column size in Figure 14. For Q1.x-Q3.x, the column sizes are similar, but MapD GPU mode outperforms Hyper with Q1.x query group and spends longer time for Q2.x and Q3.x query groups. The hot data size in GPU memory is not the only dominate factor.

The performance difference lies in the amount of joins. MapD GPU mode uses operator-at-a-time model like MonetDB, this processing model suffers from materialization overhead especially for multiple join operations. Q2.x-Q4.x involves star-join with 4 to 5 tables, and may produce large intermediate memory consumption that exceeds the GPU memory size.

According to the experimental results, MapD seems to be adaptive to queries with less joins, but MapD also seems to be non-sensitive to selectivity with similar execution time for different selectivity queries in Q1.x.

## 5.5 Relational operation performance

For detailed performance analysis, we focus on the major relational operations in OLAP workloads. We study the performance of join, star-join and grouping&aggregation operations which majorly dominate the OLAP performance.

(1) Join performance

For MapD and Hyper, we use SQL commands to simulate joins between fact table and specified dimension table, a sample SQL command is shown as:

*select count(*) from lineorder inner join date on lo_orderdate = d_datekey;*

We also use AIR algorithm from [25], NPO and PRO algorithms from [8] as candidate join algorithms.

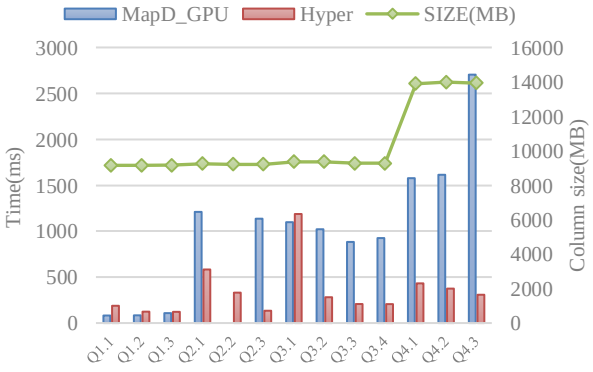



**Figure 14. Performance comparison with column size.**

**Table 2. Join performance for SSB.**

| Join/ms | LO⋈D | LO⋈ S | LO⋈ P | LO⋈C |
|---|---|---|---|---|
| **MapD CPU mode** | 92 | 271 | 290 | 296 |
| **MapD GPU mode** | 48 | 107 | 203 | 87 |
| **Hyper** | 158 | 349 | 452 | 567 |
| **AIR** | 79 | 81 | 141 | 145 |
| **AIR GPU** | 77 | 122 | 127 | 180 |
| **NPO** | 205 | 276 | 292 | 445 |
| **PRO** | 843 | 849 | 849 | 856 |

MapD's join performance is higher than Hyper for both CPU mode and GPU mode. The execution time of lineorder⋈customer is somehow strange with unreasonable short time than joins with smaller table. The MapD GPU mode takes only 1/3 to 1/2 time of Hyper spend. The MapD CPU mode is also superior to Hyper.

From state-of-the-art researches of in-memory join algorithms, we compare the representative join algorithms of AIR, NPO and PRO. NPO outperforms PRO because Intel Xeon E5-2699 v4 has 55 MB L3 cache while the biggest hash table size of customer table is smaller than L3 cache size, so that NPO can perform an in-cache hash join with higher performance than PRO. AIR uses the efficient vector and vector referencing operation to take the places of hash table and hash probing, both cache locality and code efficiency are improved. For K80 GPU, the join inputs can be horizontally partitioned to two GPU kernels for parallel processing, the performance is similar to the 44-core CPU. We use a hardware-oblivious design for AIR GPU algorithm by referencing vector from global memory no matter whether the vector size is smaller than shared memory size. Actually, the hash tables of *date* and *supplier* tables can be held in K80's 128KB shared memory. MapD GPU mode proves higher performance than other joins in lineorder⋈date and lineorder⋈supplier.

(2) Star-join performance

For MapD and Hyper, we use the following SQL commands to simulate star-join.

*select count(*) from lineorder inner join linedate on lo_orderdate=d_datekey;*

*select count(*) from lineorder inner join linedate on lo_orderdate=d_datekey inner join supplier on lo_suppkey=s_suppkey;*

*select count(*) from lineorder inner join linedate on lo_orderdate=d_datekey inner join supplier on lo_suppkey=s_suppkey inner join part on lo_partkey=p_partkey;*

*select count(*) from lineorder inner join linedate on lo_orderdate=d_datekey inner join supplier on lo_suppkey=s_suppkey inner join part on lo_partkey=p_partkey inner join customer on lo_custkey=c_custkey;*

For AIR algorithm, we use an additional vector index column as shared filter and result column. The AIR GPU algorithm simply uses global memory for vector referencing without further optimizations such as loading vector in shared memory, we hope the hardware-oblivious algorithm design can simplify database implementation with satisfied performance.

Figure 15 shows the star-join performance with SSB dataset(SF=100) for candidate join algorithms. MapD CPU mode outperforms Hyper in all 4 cases, and Hyper's star-join performance decreases as amount of joined tables increases while MapD CPU mode increase much slower. MapD GPU mode outperforms MapD CPU mode and Hyper dramatically, the

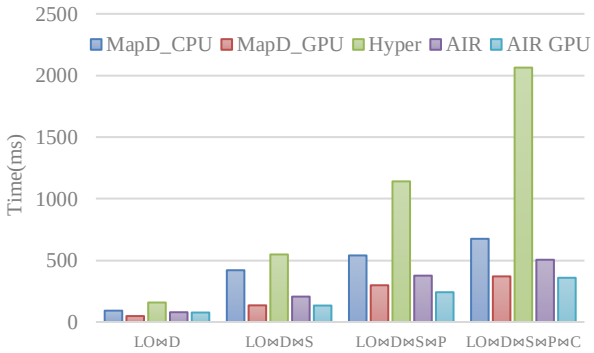



**Figure 15. Star-join performance with SSB(SF=100).**

experimental results show that the GPU mode achieves higher performance improvements than CPU mode. The AIR GPU star-join algorithm slightly outperforms MapD GPU mode, which indicates two conclusions: first, deeply optimized join algorithm of MapD GPU mode achieves significant performance gains; second, by using simple hardware-oblivious join algorithm design and OLAP domain knowledge customized AIR algorithm, the GPU star-join can also achieve good performance.

(3) Grouping and aggregation performance

The grouping and aggregation operation is the last operation in OLAP query plan tree. Vector grouping in Figure 9 can be modeled as ideal implementation for grouping and aggregation operation by compressing multiple grouping attributes into single vector.

We simulate vector grouping with one vector index column and two measure columns for aggregation. For MapD, we use the following SQL template command to simulate vector grouping operation. *lo_ordertotalprice* column is used to produce various cardinality groups by *mod*() function, and we alter the group value to simulate different grouping cardinalities. We add "limit 1" clause to minimize results printing time.

*select sum(lo_quantity*lo_tax), mod(lo_ordertotalprice, 512)*

*from lineorder*

*group by mod(lo_ordertotalprice, 512) limit 1;*

For Hyper, we use "%" instead of *mod*() function to generate specified groups. We also use "top 1" clause to minimize results printing time.

*select top 1 sum(lo_quantity*lo_tax), lo_ordertotalprice%512*

*from lineorder*

*group by lo_ordertotalprice%512;*

The grouping&aggregation benchmark(G&AB) of SQL operation can be defined as:

*select sum(Agg_expression), mod(seed_column, cardinality)*

*from T*

*group by mod(seed_column, cardinality) limit 1;*

The *seed_column* is a large cardinality *integer* column which is used to produce group IDs according to specified *cardinality* of GROUP-BY clause, the *Agg_expression* clause is manually set to simulation specified aggregation expressions. The G&AB can evaluate how the SQL engine perform grouping&aggregation operation with different grouping cardinalities, and the evaluation

**Table 3. Grouping&aggregation performance (ms)**

| Groups | MapD CPU mode | MapD GPU mode | Hyper | Vecor Grouping | | Throughput(GB/s) | |
|---|---|---|---|---|---|---|---|
| | | | | *VecGroup* | *SVecGroup* | *VecGroup* | *SVecGroup* |
| **32** | 1699 | 625 | 177 | ***122*** | 32730 | ***55.0*** | 0.2 |
| **64** | 1678 | 602 | 176 | ***121*** | 13307 | ***55.4*** | 0.5 |
| **128** | 1817 | 609 | 359 | ***122*** | 8329 | ***55.0*** | 0.8 |
| **256** | 1756 | 609 | 722 | ***123*** | 5406 | ***54.5*** | 1.2 |
| **512** | 1806 | 631 | 804 | ***122*** | 3433 | ***55.0*** | 2.0 |
| **1024** | 1790 | 642 | 946 | ***123*** | 2500 | ***54.5*** | 2.7 |
| **2048** | 1850 | 666 | 964 | ***123*** | 1739 | ***54.5*** | 3.9 |
| **4096** | 1870 | 705 | 978 | ***123*** | 1297 | ***54.5*** | 5.2 |
| **8192** | 1981 | 755 | 988 | ***130*** | 1011 | ***51.6*** | 6.6 |
| **16384** | 2208 | 818 | 991 | ***142*** | 891 | ***47.2*** | 7.5 |
| **32768** | 2360 | 870 | 1315 | ***180*** | 852 | ***37.3*** | 7.9 |
| **65536** | 2400 | 886 | 1429 | ***199*** | 830 | ***33.7*** | 8.1 |
| **131072** | 2725 | 946 | 1444 | ***256*** | 796 | ***26.2*** | 8.4 |
| **262144** | 2881 | 993 | 1449 | ***680*** | 787 | 9.9 | 8.5 |
| **524288** | 3230 | 1035 | 1433 | 1139 | ***743*** | 5.9 | 9.0 |
| **1048576** | 3818 | 1107 | 1451 | 1268 | ***704*** | 5.3 | 9.5 |
| **2097152** | 4225 | 1258 | 1504 | 1567 | ***689*** | 4.3 | 9.7 |
| **4194304** | 5346 | 1561 | 1673 | 1888 | ***834*** | 3.6 | 8.0 |
| **8388608** | 7015 | 1789 | 1785 | 2468 | ***1391*** | 2.7 | 4.8 |
| **16777216** | 10530 | 2604 | 1888 | 3456 | ***1743*** | 1.9 | 3.8 |
| **33554432** | 16658 | 3968 | 1961 | 5295 | ***1927*** | 1.3 | 3.5 |
| **67108864** | 17406 | 4292 | ***1978*** | -- | 2102 | | 3.2 |

results can further discover the relationship of cardinality, cache size and performance.

We also develop vector grouping algorithm family to evaluate the grouping&aggregation performance. We develop algorithms inside the open-source code of [8], using the same programming style and *latch* mechanism to illustrate the benchmark performance. The *VecGroup*() function is designed for vector grouping algorithm with private vector in each aggregation thread, the *SVecGroup*() function uses shard vector for all the parallel aggregation threads with concurrent control mechanism.

We vary the groups from $2^5$(32) to $2^{26}$(67108864), the execution time is shown in Table 3. The underlined value in each row is the shortest execution time for all the candidate operations. We first focus on MapD and Hyper, and we find that MapD CPU mode is 2-4 times slower than MapD GPU mode, MapD CPU mode is also dramatically slower than Hyper especially for very small and very large groups (up to 8-9 times slower than Hyper), MapD GPU mode outperforms Hyper for moderate groups(256 to 4194304) and is slower than Hyper for very small(smaller than 128) and very large(larger than 8388608) groups.

For vector grouping algorithms, the performance is dominated by ratio of private vector size to cache size. *VecGroup* function outperforms *SVecGroup* function for cache fit vectors, e.g., the vector size of 262144 groups is about 1MB which is close to 1/2 L3 cache slice(L3 cache slice size is 2.5 MB) of each thread(one core has two physical threads). When private vector size exceeds thread share of L3 cache slice, the *SVecGroup* function with shared vector begins to outperform *VecGroup* function. Moreover, *VecGroup* algorithm spends more memory for private vectors of each thread than single shared vector in *SVecGroup* function, the memory runs out for 67108864 groups. *SVecGroup* function is extremely slow for small groups smaller than 1024 due to heavy concurrent updating on shared vector. As vector size grows, the concurrent conflict reduces and the execution time keeps reducing to the minimal execution time with 2097152 groups. When vector size keeps growing, the execution time of *SVecGroup* function increases for large vector size and produces more cache misses.

Therefore, the vector grouping algorithm family outperforms the leading databases both on CPU and GPU platforms. The performance improvements achieve 695% and 481% against Hyper and MapD. The average grouping and aggregation execution time of MapD CPU mode is 9.9 times of vector grouping algorithm. The average grouping and aggregation execution time of MapD GPU mode is 3.77 times of vector grouping algorithm. The average grouping and aggregation execution time of Hyper is 4.29 times of vector grouping algorithm. Against MapD, the vector grouping operation achieves 2.7-15.55 times speedup over CPU mode and 1.39-5.81 times speedup over GPU mode.

As real-time OLAP is commonly applied for interactive analysis processing with moderate groups to be understand by users, performance of moderate group size is more representative. For general scenarios, *VecGroup* function is the best choice for lower than 256 K groups, for even more groups, *SVecGroup* function can also provide good performance.

For MapD GPU aggregation, we consecutively run the SQL statements several times and select the minimal run times as GPU resident aggregation time. For benchmark evaluation, the GPU mode may transfer measure columns through PCIe channel to GPU

memory for aggregation, the total time includes data transfer time and aggregation time. For example, the PCIe transfer time of 3 columns under SSB with SF=100 needs 450 *ms*, and MapD GPU mode has to spend more time than Hyper in all grouping cardinality evaluations. We design a baseline rule for GPU database optimizer on whether grouping&aggregation operation should be assigned to GPU in this paper. The rule is dominated by PCIe bandwidth and grouping&aggregation operation throughput:

$$\text{platform} = \begin{cases} GPU & if\ BandWith_{PCIe} > ThroughPut_{G\&A} \\ CPU & if\ BandWith_{PCIe} < ThroughPut_{G\&A} \end{cases}$$

In table 2, we calculate the throughput (GB/s) of grouping&aggregation operation *VecGroup*() and *SVecGroup*() functions, we see that *VecGroup*() function with cardinalities under 131072 is higher than PCIe 3.0 transfer bandwidth(16GB/s for single direction), and we can draw conclusion that grouping&aggregation operation with moderate cardinalities should be assigned to CPU instead of GPU.

## 5.6 Vector grouping performance

To comprehensively evaluate the vector grouping performance, we design the cache-conscious and selectivity-aware experiments. For cache-conscious experiments, we compare the private vector grouping and shared vector grouping algorithms to discover how cache size influences the vector grouping performance. For selectivity-aware experiments, we use fixed length vector and compressed vector for grouping with different selectivities.

**(1) Cache-conscious vector grouping evaluations**

We use an INT type vector as aggregator, the vector size is used as parameter to set vector sizes which equal to different proportions of L1 cache size, L2 cache size, L3 cache slice and LLC size.

We develop the vector grouping algorithm in the open-source code from [8], and use the same *latch* mechanism for shared vector aggregation. The CYPER-PER-TUPLE is used for performance evaluation, the lower CYPER-PER-TUPLE is the faster the vector grouping algorithm runs.

Figure 16 shows the vector grouping performance with private vector and shared vector, the vector size varies from 10% of L1 cache size(32KB, 819 groups) to 200% L1 cache size(64KB, 16384 groups). We can see that private vector grouping outperforms shard vector grouping dramatically, shared vector grouping suffers from concurrent aggregation updating overhead while private vector grouping is much efficient with updating on independent private vectors. The maximal performance gap between them is 22 times large. As the shared vector size increases, the concurrent conflicts on each cell is reduced and the performance gradually increases.

Figure 17 illustrates that the private vector grouping still outperforms shared vector grouping within L2 cache size boundary. As group member grows, the performance of private vector gradually drops while the shared vector increases. However, as vector size increases large enough, the private vector grouping gradually runs slower than shared vector grouping. When vector size exceeds 60% of L3 cache slice(2.5MB, 393216 groups), shared vector grouping begins outperforming private vector grouping as shown in figure 18. For this threshold, two threads of one core shares the whole L3 cache slice, more cache misses are produced as vector size grows beyond this threshold. As we increase the vector size in figure 19 from 100% L3 cache slice size to 2000% L3 cache slice size, we find that the CYCLE-PER-TUPLE of shared vector and private vector both keep increasing for big vector

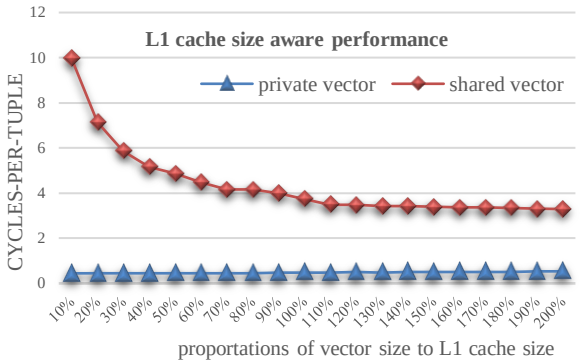



**Figure 16. L1 cache size aware vector grouping performance.**

size produces more cache misses. Figure 20 shows that the performance gap between shared vector grouping and private vector grouping becomes larger when vector size exceeds the whole LLC size(55MB) because more threads and larger vector size of private vector grouping produces more cache misses than single shared vector with concurrent updating.

**(2) Selectivity-aware vector grouping evaluations**

When query selectivity is low, the vector is sparse with many NULL cells. The compressed vector grouping algorithm uses compressed vector instead of fixed length vector with (OID, VAULE) pairs, the scan overhead on vector is minimized.

Figure 21 shows the performance of vector grouping and compressed vector grouping performance. For compressed vector grouping, the CYCLE-PER-TUPLE drops linearly as selectivity drops. For high selectivity, compressed vector is bigger than non-compress vector, and compressed vector grouping is faster than non-compress vector grouping when selectivity is lower than 80%. The execution time of non-compress vector grouping doesn't drop linear like compressed vector grouping, the run time of moderate selectivity drops slower than high or low selectivity because the hardware branch prediction mechanism works better in high or low selectivity scenarios.

When selectivity varies from 70% to 30%, the vector grouping run time drops from 20% to 60%, when selectivity is 1%, run time drops about 250%, when selectivity drops from 0.1% to 0.001%, run time drops about 10 times. The selectivity of SSB various from 3.4% to 0. 000076%, the compressed vector grouping is well-suited for SSB.

## 5.7 GPU OLAP acceleration

MapD outperforms Hyper for major OLAP operators such as join, star-join and grouping&aggregation, while Hyper outperforms MapD for the benchmark evaluation. Considering the query involving columns size and the weakness of operator-at-a-time processing model, we owe to performance decreasing to materialization overhead which causes memory transfer cost between main-memory and GPU memory through low bandwidth PCIe channel. How to improve MapD's performance in Benchmark testing to what MapD should be is an important issue in this paper.

*Vectorized processing* model can optimize materialization overhead by materialize intermediate columns inside L1 cache. The vector length can be modeled as:

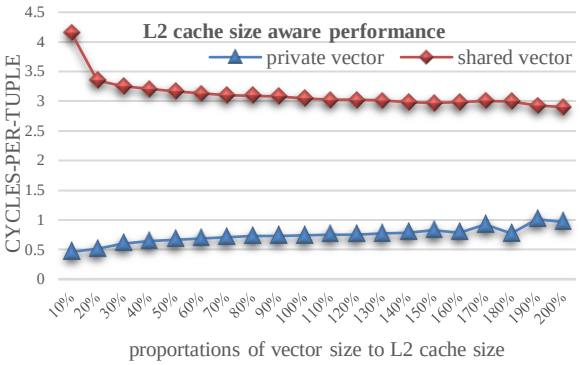



**Figure 17. L2 cache size aware vector grouping performance.**

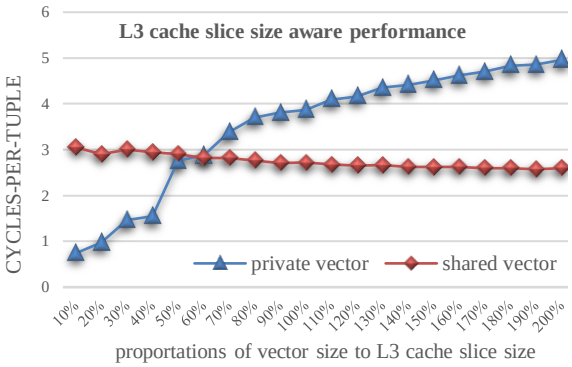



**Figure 18. L3 cache slice size aware vector grouping performance.**

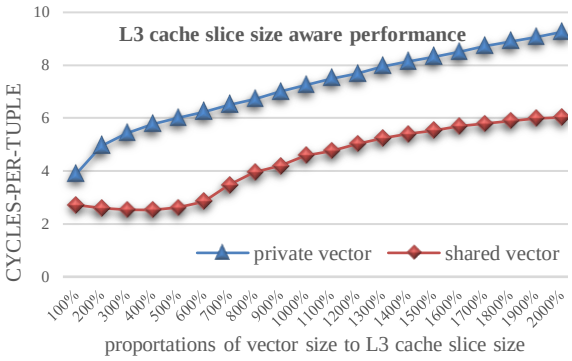



**Figure 19. L3 cache slice size aware vector grouping performance.**

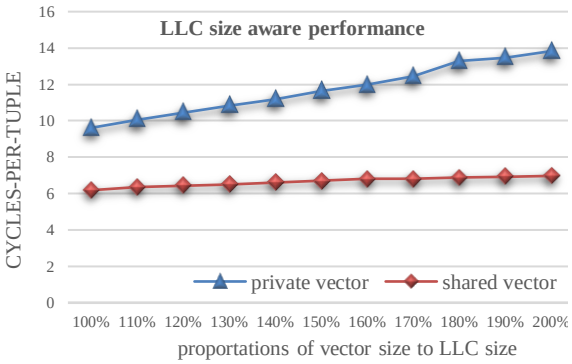



**Figure 20. LLC size aware vector grouping performance.**

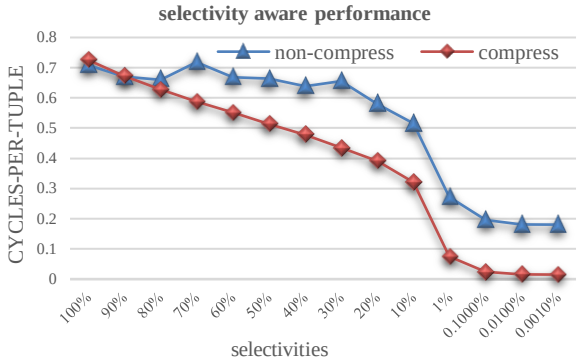



**Figure 21. Selectivity aware vector grouping performance.**

$$Vec_{len} = \frac{L1\ cache\ size}{NumberOfColumns}$$

For CPU architecture, each core has private 64KB L1 cache, so each thread can perform *vectorized processing* independently. For GPU architecture, each SMX has hundreds of core(e.g. 192 core inside one SMX in K80) with the single shared memory(e.g., 128 KB shared memory in K80), the *vectorized processing* model is more difficult to be implemented for GPUs than commonly adopted operator-at-a-time processing model.

To minimize materialization overhead is important for operator-at-a-time processing model in GPUs. The AIR star-join and vector grouping are designed to minimize materialization overhead by pre-compressing grouping attributes into small encoding values and use the multidimensional cube model to integrate multiple grouping codes into single shared vector index column. So that, an additional vector index column is used as shared intermediate materialization column for all the OLAP queries. Moreover, the vector grouping is also efficient for using vector offset address instead of long grouping attributes for aggregation without hash probing overhead.

Considering the OLAP workloads e.g., SSB, we can further evaluate the efficiency of GPU database or GPU acceleration mechanisms. Consequently, we further evaluate GPU acceleration mechanism with SSB workload features.

Table 4 illustrates how to accelerate OLAP performance with OLAP domain knowledge customized implementations. Q1.x majorly involves selection operation in big fact table, we mainly focus on acceleration star-join in this paper and neglect Q1.x group. We use fixed length vector index as shared intermediate materialization column for all the queries, the vector index is defined as *short* type with 2 bytes, the transfer time through PCIe 3.0 channel(16GB/s) is about 73 *ms*. For low selectivities queries, the vector index can be further compressed. As compressing vector index needs more optimizations for dynamically allocating GPU memory, we will study it in our future work and in this paper simply assumed that the compressed vector index transfer time is at most 73 *ms*. With compressed vector grouping described in Figure 4, the CVecGroup operation is very efficient for the low selectivities.

The GPU OLAP acceleration need an additional module to rewritten OLAP query to dimension vectors. We have studied how to generate dimension vectors by customized function with average time 30 *ms* for SSB(SF=100)[25], and how to generate dimension vectors by SQL statements with average time 68 *ms* for

**Table 4. OLAP acceleration for SSB.**

| Query/ ms | AIR star-join for CPU | AIR star-join for GPU | Vector index transfer | CVec Group | OLAP acceleration for CPU | OLAP acceleration for GPU |
|---|---|---|---|---|---|---|
| Q2.1 | 284 | 226 | 73 | 14 | 298 | 313 |
| Q2.2 | 268 | 198 | 73 | 5 | 273 | 276 |
| Q2.3 | 234 | 217 | 73 | 2 | 236 | 292 |
| Q3.1 | 385 | 267 | 73 | 72 | 457 | 412 |
| Q3.2 | 289 | 251 | 73 | 5 | 294 | 329 |
| Q3.3 | 199 | 193 | 73 | 2 | 201 | 268 |
| Q3.4 | 174 | 198 | 73 | 2 | 176 | 274 |
| Q4.1 | 422 | 314 | 73 | 46 | 468 | 433 |
| Q4.2 | 376 | 307 | 73 | 9 | 385 | 389 |
| Q4.3 | 360 | 276 | 73 | 2 | 362 | 351 |
| Average time | 299 | 245 | 73 | 16 | 315 | 334 |

SSB(SF=100) in Hyper. Considering the latest NVLink technique[3], the vector index transfer time with NVLink's 300GB/s bandwidth can decrease average 73 *ms* to 4 *ms*. So we summarize the possible GPU OLAP acceleration techniques in Table 5 to evaluate how to achieve the best performance gains with GPUs.

In Table 5, we focus on the queries with star-join operations. We find that MapD CPU mode has close performance to Actian Vector with similar *vectorized processing* optimization, MapD GPU mode also has close performance to MonetDB with similar operator-at-a-time processing model, and Hyper proves to be 2 or 3 times faster than other databases. A straightforward issue is how to further accelerate OLAP performance with GPUs against the Hyper's high performance.

In "OLAP acceleration model" part of Table 5, we layout the candidate star-join, vector transfer, vector grouping, Dim vector generating execution time to evaluate current and ideal OLAP acceleration model performance. Current OLAP acceleration model uses PCIe 3.0 transfer time and SQL Dim vector generating time, the GPU acceleration execution time is equal to Hyper and CPU acceleration execution is faster than Hyper. The ideal OLAP acceleration model simulates using NVLink transfer time and customized function for Dim vector generating, both OLAP acceleration by CPU and GPU are faster than Hyper, and GPU OLAP acceleration is even faster for the star-join performance improvements by GPUs. The experimental results show that a general-purpose relational engine accelerated by GPUs like MapD may not bring significant performance gains even when the working data can be completely held in GPU memory. However, an OLAP domain knowledge customized GPU OLAP acceleration can efficiently improve OLAP performance by simply accelerating star-join with GPUs and using efficient vector grouping with CPUs. Therefore, the proposed OLAP acceleration approach can achieve 2.23 times faster than MapD CPU mode and 3.58 times faster than MapD GPU mode with SSB workload (SF=100).

Nowadays GPU servers are commonly configured with 8-16 GPUs for scale-out computing, so that the GPU OLAP acceleration model performance can be further improved by partitioning star-join workloads to more GPUs for parallel computing. As GPU OLAP acceleration spend around 83% execution time in GPU star-join stage, the scalability of GPUs can dramatically speedup OLAP performance with near linear speedup ratio of GPU star-join.

Finally, we summarize the main empirical findings as follows:

(1) State-of-the-art GPU databases (e.g. MapD) suffer from data movement overhead and low GPU memory utilization.

(2) The grouping and aggregation operation differs remarkably with different implementations. The proposed vector grouping algorithm achieves 2.7-15.55 times speedup over MapD CPU version and 1.39-5.81 times speedup over MapD GPU version with various grouping cardinalities.

(3) The overall OLAP performance of proposed approaches may be 2.23 times faster than MapD CPU version and 3.58 times faster than MapD GPU version for the Star Schema Benchmark (SSB) with scale factor 100.

# 6. RELATED WORK

GPU databases emerge to become main-stream high performance databases in recent years. In the latest TOP 500 list[30], there are 5 systems are configured with GPUs in top 7 systems. The industrial also pushed forward GPU databases like MapD[1], Kinetica[4],

**Table 5. GPU OLAP acceleration for SSB.**

| Query/ ms | MapD CPU | MapD GPU | Hyper | Monet DB | Vector | OLAP Acceleration model | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | CPU star-join | GPU star-join | vector index trans. by PCIe | vector index trans. by NVLink | CVec Group | GenDim by SQL | GenDim by function | OLAP acceleration by CPU | OLAP acceleration by GPU | Ideal OLAP acceleration by CPU | Ideal OLAP acceleration by GPU |
| **Q2.1** | 732 | 1210 | 583 | 1200 | 775 | 284 | 226 | 76 | 4 | 14 | 54 | 24 | 352 | 370 | 323 | 269 |
| **Q2.2** | | | 331 | 1100 | 547 | 268 | 198 | 76 | 4 | 5 | 59 | 24 | 332 | 338 | 297 | 231 |
| **Q2.3** | 728 | 1136 | 134 | 649 | 342 | 234 | 217 | 76 | 4 | 2 | 26 | 8 | 262 | 321 | 245 | 232 |
| **Q3.1** | 842 | 1098 | 1187 | 1300 | 1615 | 385 | 267 | 76 | 4 | 72 | 85 | 63 | 542 | 500 | 520 | 406 |
| **Q3.2** | 822 | 1021 | 281 | 1000 | 996 | 289 | 251 | 76 | 4 | 5 | 91 | 28 | 385 | 423 | 322 | 288 |
| **Q3.3** | 810 | 883 | 207 | 574 | 540 | 199 | 193 | 76 | 4 | 2 | 73 | 20 | 274 | 344 | 221 | 219 |
| **Q3.4** | 889 | 925 | 206 | 535 | 495 | 174 | 198 | 76 | 4 | 2 | 73 | 22 | 249 | 350 | 197 | 226 |
| **Q4.1** | 962 | 1577 | 432 | 1900 | 1795 | 422 | 314 | 76 | 4 | 46 | 81 | 44 | 549 | 517 | 513 | 408 |
| **Q4.2** | 950 | 1614 | 375 | 1500 | 1462 | 376 | 307 | 76 | 4 | 9 | 65 | 51 | 450 | 457 | 436 | 371 |
| **Q4.3** | 1836 | 2704 | 308 | 1000 | 981 | 360 | 276 | 76 | 4 | 2 | 69 | 15 | 431 | 424 | 377 | 297 |
| **Aver time** | ***952*** | ***1352*** | ***404*** | 1076 | 955 | 299 | 245 | 76 | 4 | 16 | 68 | 30 | 383 | 404 | ***345*** | ***295*** |

SQREAM[5], Zilliz[6] and BlazingDB[31]. Due to the significant difference between CPU and GPU, the sophisticated optimization techniques of in-memory databases cannot be directly implemented by GPU databases, and also cannot guarantee to be more efficient in GPU than in CPU.

The GPU database's performance is dominated by multiple factors, e.g., making hot data GPU memory resident, evaluating GPU friendly operators from CPU friendly operators, using cost model to distribute operations between GPU and CPU[23]. The common GPU database query processing model is operator-at-a-time model[22] which is proved to be less efficient than *vectorized processing* model[24] and JIT compilation model[20] for in-memory databases. Moreover, operator-at-a-time model suffers from large materialization overhead which can be even worse in GPU databases for the limited GPU memory size. MapD is the representative GPU database with *vectorized processing*, JIT compilation and optimization of keeping hot data in GPU memory as much as possible, but we still find that MapD are not proved to be superior to in-memory database Hyper with OLAP benchmark (SSB) even when hot data size is smaller than GPU memory size. So that, GPU databases may not be the only answer to acceleration OLAP performance.

The general-purpose relational processing model is not the best-suited solution for OLAP workloads. The pipelined processing model mixes computing-intensive and data-intensive workloads together which causes the most costly join overhead only occupy 10%–15% share of the total runtime[11] which decrease the efficiency of GPU acceleration. [25] proposed a tailored array-store model to simplify query processing model in which traditional joins are replaced by array index referencing (AIR) from foreign key to dimension vector. The group-by clause is also modeled as multidimensional array aggregation instead of hash aggregation. By these optimizations, the star-join is limited in moderate foreign key columns with more than 80% execution time.

[26] generalized the customized OLAP model as fusing MOLAP and ROLAP model together. The OLAP is clearly defined with 3 stages: dimension mapping, multidimensional filtering and aggregation. The vector index is proposed to implement MOLAP model inside ROLAP model, the vector index maintenance mechanism is also discussed for updating.

This paper studied GPU database MapD's performance characteristics for OLAP workloads, discovering the advantages and disadvantages aspects of MapD to locate critical bottleneck of MapD. Combined with MapD's performance features and customized OLAP model, we also discussed the GPU OLAP acceleration model as OLAP domain knowledge customized model for GPU OLAP engine.

# 7. CONCLUSIONS

In this paper we have studied how to optimize GPU database framework by analyzing and optimizing the dominated OLAP operations and developing the platform-adaptive operations.

In particular, we first designed the 3-stage-computing model as tailored OLAP framework. The tailored OLAP framework separates the computing-intensive workload and data-intensive workload from the tightly coupled query processing workload, and each stage can be assigned to specified processing model, which makes OLAP scalable for the emerging hybrid processing platform. Moreover, the 3-stage-computing model extends the relational engine to be an open computing framework, and each stage can be accelerated by specified technique using vector index as loose coupled intermediator data structure. The vector oriented algorithm designs simplify the star-join and grouping algorithms by using simple vector instead of hash table. Our experimental results show that the tailored vector grouping optimization minimizes the costly transfer overhead to achieve better performance. In our future work, we plan to develop the asynchronous hybrid CPU-GPU platform query processing model for concurrent queries to maximize utilizations for both CPUs and GPUs.

# 8. ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (61772533, 61732014), the Natural Science Foundation of Beijing (4192066) and Academy of Finland, Finland (310321).

# 9. REFERENCES

[1] MapD is the Extreme Analytics Platform. (2018, Feb.). [Online]. Available: https://www.mapd.com/

[2] Thomas Neumann, Viktor Leis: Compiling Database Queries into Machine Code. IEEE Data Eng. Bull. 37(1): 3-11 (2014)

[3] HGX-2 Fuses HPC and AI Computing Architectures. (2018, Jun.). [Online]. Available: https://devblogs.nvidia.com/hgx-2-fuses-ai-computing/

[4] Kinetica is the insight engine for the Extreme Data Economy. (2018, Jan.). [Online]. Available: https://www.kinetica.com/

[5] SQream DB is the GPU Data Warehouse for massive data. (2018, Feb.). [Online]. Available: https://sqream.com/

[6] FASTEST, SMARTEST-AI-Oriented Data Processing Platform. (2018, Apr.). [Online]. Available: http://zilliz.com/

[7] PG-Strom - Master development repository. (2017, Sep.). [Online]. Available: https://github.com/heterodb/pg-strom

[8] Cagri Balkesen, Jens Teubner, Gustavo Alonso, M. Tamer Özsu: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. ICDE 2013: 362-373

[9] Cagri Balkesen, Gustavo Alonso, Jens Teubner, M. Tamer Özsu: Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. PVLDB 7(1): 85-96 (2013)

[10] Stefan Richter, Victor Alvarez, Jens Dittrich: A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. PVLDB 9(3): 96-107 (2015)

[11] Stefan Schuh, Xiao Chen, Jens Dittrich: An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. SIGMOD Conference 2016: 1961-1976

[12] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, Rick Siow Mong Goh: Efficient Query Processing on Many-core Architectures: A Case Study with Intel Xeon Phi Processor. SIGMOD Conference 2016: 2081-2084

[13] Xuntao Cheng, Bingsheng He, Xiaoli Du, Chiew Tong Lau: A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. CIKM 2017: 657-666

[14] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, Pedro V. Sander: Relational joins on graphics processors. SIGMOD Conference 2008: 511-524

[15] Ran Rui, Hao Li, Yi-Cheng Tu: Join algorithms on GPUs: A revisit after seven years. Big Data 2015: 2541-2550

[16] Jiong He, Mian Lu, Bingsheng He: Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. PVLDB 6(10): 889-900 (2013)

[17] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, Vassilis J. Tsotras: FPGA-based Multithreading for In-Memory Hash Joins. CIDR 2015

[18] Kaan Kara, Jana Giceva, Gustavo Alonso:FPGA-based Data Partitioning. SIGMOD Conference 2017: 433-445

[19] MapD Technical Whitepaper. The world's fastest platform for data exploration. (2018, Sep.). [Online]. Available: http://go3.mapd.com/resources/whitepapers/mapd/lp

[20] Juliusz Sompolski, Marcin Zukowski, Peter A. Boncz: Vectorization vs. compilation in query execution. DaMoN 2011: 33-40

[21] Emily Furst, Mark Oskin, Bill Howe: Profiling a GPU database implementation: a holistic view of GPU resource utilization on TPC-H queries. DaMoN 2017: 3:1-3:6

[22] Peter A. Boncz, Martin L. Kersten, Stefan Manegold: Breaking the memory wall in MonetDB. Commun. ACM 51(12): 77-85 (2008)

[23] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, Gunter Saake: GPU-Accelerated Database Systems: Survey and Open Challenges. Trans. Large-Scale Data- and Knowledge-Centered Systems 15: 1-35 (2014)

[24] Marcin Zukowski, Peter A. Boncz, Niels Nes, Sándor Héman: MonetDB/X100 - A DBMS In The CPU Cache. IEEE Data Eng. Bull. 28(2): 17-22 (2005)

[25] Yansong Zhang, Xuan Zhou, Ying Zhang, Yu Zhang, Mingchuan Su, Shan Wang: Virtual Denormalization via Array Index Reference for Main Memory OLAP. IEEE Trans. Knowl. Data Eng. 28(4): 1061-1074 (2016)

[26] Yansong Zhang, Yu Zhang, Shan Wang, Jiaheng Lu. Fusion OLAP: Fusing the Pros of MOLAP and ROLAP Together for In-memory OLAP. DOI: 10.1109/TKDE.2018.2867522. [Online]. Available: https://ieeexplore.ieee.org/document/8449096

[27] HyPer—A Hybrid OLTP&OLAP High Performance DBMS. (2015, 1172 Feb.). [Online]. Available: http://hyper-db.com/

[28] (2017, Apr.). [Online]. Available: https://github.com/mapd

[29] (2016, Sep.). [Online]. Available: https://github.com/heterodb/pg-strom

[30] JUNE 2018. (2018, Jun.). [Online]. Available: https://www.top500.org/lists/2018/06/

[31] GPU-Accelerated Analytics on your Data Lake. (2018, Oct.). [Online]. Available: https://blazingdb.com/

[32] Daniel J. Abadi, Samuel Madden, Nabil Hachem: Column-stores vs. row-stores: how different are they really? [C].SIGMOD Conference 2008: 967-980