Advanced database systems

The Survey of GPU-Accelerated Database Systems

THE UNIVERSITY OF

MELBOURNE

Group 133

Abhinav Sharma, 1009225
sharma.a1@student.unimelb.edu.au

Rohit Kumar Gupta, 1023418
rohitkumarg@student.unimelb.edu.au

Shun-Cheng Tsai, 965062
shunchengt@student.unimelb.edu.au

Hyesoo Kim, 881330
hyesook@student.unimelb.edu.au

Abstract

GPUs can perform batch processing of considerable chunks in parallel, providing extreme performance over traditional CPU based system. Their usage has shifted drastically from exclusive video processing to generic parallel processing. In the past decade, the need for better processors and computing power has increased due to technological advancements achieved by the computing industry. GPUs work as a coprocessor of CPUs and the integration of their architectures has raised many bottlenecks.

Multiple GPUs solutions have been developed by industry to tackle these bottlenecks. In this paper, we have explored the current state of GPU-accelerated database industry. We have surveyed eight such databases in the industry and compared them on eight most prominent parameters such as performance, portability, query optimization, and storage models. We have also discussed some of the significant challenges in the industry and the solutions that have been devised to manage these bottleneck.

Contents

1 Introduction 1

2 Background Considerations 2

3 The design-space of GPU-accelerated DBMS 3

4 A survey of GPU-accelerated database systems 4
- 4.1 CoGaDB . . . 4
- 4.2 GPUDB . . . 6
- 4.3 OmniSci/MapD . . . 8
- 4.4 Brytlyt . . . 10
- 4.5 SQream . . . 13
- 4.6 PGstrom . . . 15
- 4.7 OmniDB . . . 17
- 4.8 Virginian . . . 19

5 GPU-accelerated Database Systems Comparison 21
- 5.1 Functional Properties . . . 21
- 5.2 Non-Functional Properties . . . 24

6 Open Challenges and Research Questions 25

7 Conclusion 27

1 Introduction

Moore's law states that the processor speeds or overall processing power for computers would double every two years[1]. Due to technological advancements in transistor speed and energy scaling, microprocessors performance had grown consistently for decades[1]. However, decline transistor-speed growth and physical limitations of energy have generated new challenges for performance scaling, and as a result, the growth rate has declined. With these limiters of performance, research has moved to designs that utilize the large-scale parallelism, heterogeneous cores, and accelerators to deliver performance and energy efficiency[1]. The software-hardware partnership is being explored to achieve efficient data orchestrations to achieve energy-proportional computing[1]

GPUs have evolved into notably powerful and flexible processors, and their architecture provides enormous memory bandwidth and computational power. These GPUs now have programmable units which can support vector operations and IEEE floating point precision[2]. This has facilitated the GPUs architectures to develop GPU based solutions for data-parallel computations. The high processing power and memory bandwidth of modern graphics cards make them a powerful platform for data-intensive applications with the capability to execute massive calculation on data in parallel [3][4]. In the Big-Data era, this capability of GPU has lead to strong demand for GPU accelerated databases.

Over the last decade, serious progress has been made by the database research community to investigate the methods to accelerate database systems integration of GPUs. Various research papers, along with performance studies, have demonstrated the true potentials of GPU integrated Databases, which has eventually led to the development of a few matured commercial GPU-accelerated database products[4]. These products are integrated at various levels with existing architectures and have been designed to act as a powerful coprocessor[4].

Despite the drastic increase in flexibility of GPUs, many applications still lie outside the scope and don't suit the GPU based solutions. For example, Word processing that requires high memory communication and cant be easily parallelized[2].

Acceleration of databases using GPUs has its own set of bottlenecks and pitfalls. The most significant hurdle is the data transfer bottleneck between CPU and GPU, which requires reducing and concealing the delays via various data management and caching models[5]. There are some challenges due to the variation in storage models and storage strategies across systems. Even the integration of GPUs with "real-world" Database Management System (DBMS) faces issues related to query processing and data structures[5].

In this paper, we have:

1. studied eight GPU-accelerated DBMSs (GDBMSs) to evaluate each of them on six non-functional and two functional parameters

2. provided a comparison of each GDBMS on individual parameters

3. discussed some of the major open challenges and solutions propositions

[1]https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf

2 Background Considerations

In this section, we provide a brief overview of the Graphics processing unit (GPU) by explaining its architecture and working. A Graphics Processing Unit (GPU) is a specialized electronic device which is initially designed with the purpose of generating output frames intended to display on some output device. It can now be used as the general computation device for performing several large computational operations in diverse multi-threaded environments, which is primarily called GPGPU[2]. These GPUs have immense processing power due to their large bandwidth and extreme speed for solving floating point operations as compared to CPUs, which is shown in Figure 1.

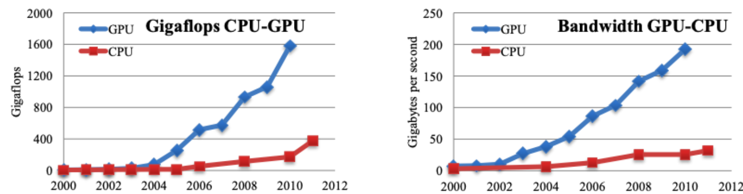

Figure 1: Comparison of performance for NVIDIA GPUs and Intel CPUs, taken from [6]

Figure 2 shows the architecture of a general modern computer system installed with a graphics card. The graphics card, which is an external device, is connected to the host system using PCI express[3] bus. The graphics card contains its device memory and typically it cannot access the host system's main memory. GPU can only interact with graphics card device memory due to which data is required to be transferred from the host's main memory to device memory using a low-bandwidth bus. The GPU itself contains several small multiprocessors which use on-chip shared memory with memory controllers and instructor decoders. Due to these several multiprocessors, GPU provides extreme multi-threaded processing environment which enables tasks to perform heavy computations in a small interval of time.

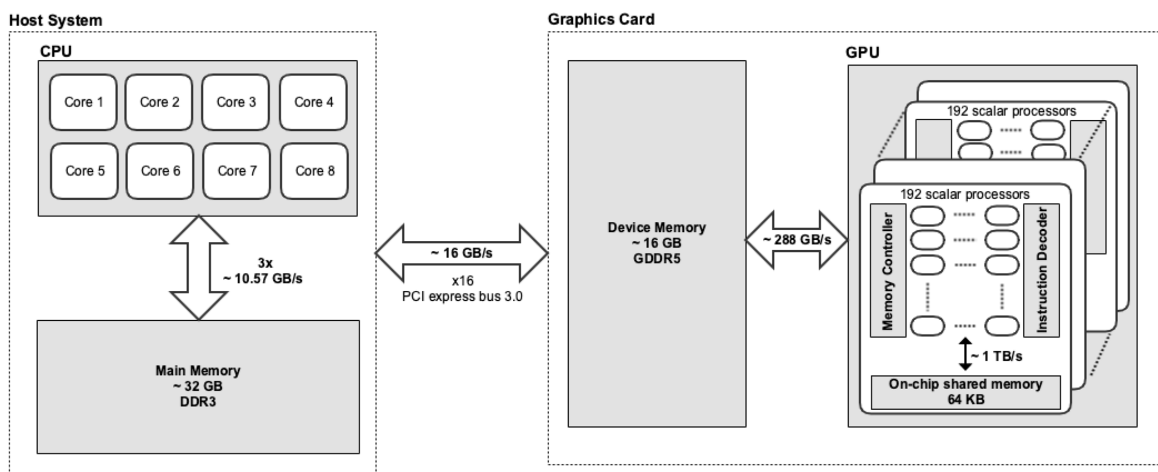

Figure 2: Exemplary architecture of a system with a graphics card, adapted from [7]

[2]General purpose graphics processing unit

[3]Peripheral Component Interconnect express(PCIe) - interface for connecting high-speed components

3 The design-space of GPU-accelerated DBMS

In this section, we will discuss the design-space of GPU-accelerated database systems. By exploring the design-space, we are trying to figure out the parameters on which we can survey and compare different database systems which support GPU acceleration.

Functional Properties. The properties which are comprised of understanding the functionalities of the system. These properties help us in providing a technical understanding of the workflows and give deep insight on how everything is managed in the system.

Storage System. A functional property which can be used to understand the underlying storage system of database systems with GPU-acceleration. In terms of the design-space, the storage system can describe the decision of GDBMS[4] of choosing a disk-based system or host's main-memory as their primary storage space.

Storage Model. After database system has decided which storage system will work as per their requirement, this property describes their decision of choosing how data is going to store in the selected storage system. These systems can store their data in either row-store or column-store format.

Processing Model. This property describes the model adopted by the database system to process SQL queries. There are several processing models which are used in the database system widely such as tuple-at-a-time, operator-at-a-time, bulk-processing, record-oriented-processing, and others.

Buffer Management. This property is used to describe the policies and strategies adopted by the GPU-accelerated database systems to manage data transfer requirements. Some system tends to create a specialized module while other manages it by implementing isolated programs which are usually executed on the host machine.

Query Placement and Optimizations. This property is used to describe how optimizations are implemented in these database systems. Query placement describes scenarios where the system is able to decide which part of the query should be executed on which device in a multi-processing environment.

Consistency and Transactions Processing. This property is used to describe how the database system manages consistency and process transactions. It also analyzes the system's ability to perform all transaction related operations such as commit, rollback and locking mechanisms such as 2-phase locking, optimistic locking, and others.

Non-Functional Properties. The properties which are used to understand non-functional aspects of the system. These properties are not used to describe the system's technicalities and framework workflows. They are used to explain the overall system's operations, different identifiers, summarizing attributes, and giving brief insights on the quality parameters.

Performance. This non-functional property helps us in reviewing the entire system's working and efficiency. It can describe the overall impact of the database system which can be compared with other similar systems for analyzing the implemented system's ability and throughput.

Portability. This non-functional property is used to understand the system's ability to be independent of hardware requirements and specifications. It is used to analyse whether the database system is hardware-oblivious, independent of the hardware provided, or hardware-aware, dependent on the hardware specification.

[4]GDBMS - GPU-accelerated database systems

4 A survey of GPU-accelerated database systems

4.1 CoGaDB

4.1.1 Overview

CoGaDB[5] is a column-oriented coprocessor-accelerated database system which effectively provides a high-performance OLAP engine[8] that makes efficient use of GPUs, CPUs or MICs (Xeon Phi) for query processing. This system is developed by the DIMA group at Berlin and the IAM group at the German Research Center for Artificial Intelligence.

CoGaDB implements a specialised cost modelling algorithm for selection of the most efficient heterogeneous processor environment using parallel query processor and *Hybrid Query Processing Engine*(HyPE)[9]. It also uses a customised query compiler called Hawk[10] for generation of specialised code for different processors which are highly optimised depending upon the query workload and data set.

4.1.2 Architecture

S Breß and others developed CoGaDB architecture to be highly efficient and modularized as shown in the top-down approach in Figure 3. As most DBMSs, CoGaDB provides SQL frontend interface which can be used to write SQL language queries and launch it to get results from the database. Then, CoGaDB's logical optimizer applies pre-defined optimization policies to generate a new query plan which is then provided to HyPE[9] for processing. The hybrid query optimizer creates an optimized physical query plan from the logical query plan which is then passed to Hawk[10] for machine code generation.

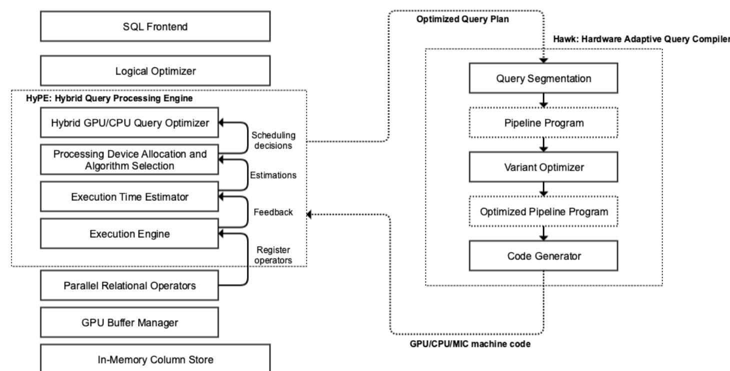

Figure 3: The architecture of CoGaDB, adapted from [10] and [11]

4.1.3 Exploring the design-space

4.1.3.1 Functional properties

Storage system

The work of He and others showed that GPU acceleration is not beneficial if data is required to be fetched from the disk[12]. Therefore, CoGaDB uses main-memory as the primary source of storage system for its working. Moreover, CoGaDB can transfer data from main memory to GPU very efficiently, leading to an immense increase in the GPU acceleration.

[5]http://cogadb.dfki.de/

Storage model
CoGaDB needs an efficient storage model to work with the requirement of loading the entire database in main memory during startup. Taking this requirement into consideration, CoGaDB chooses a columnar data layout as the storage model because of higher compression rates which can save immense memory and gives provision of storing data more efficiently due to the columnar nature.

Processing model
CoGaGB is designed with the objective of using all available processors depending upon the operators to improve query processing and performance. Thus, CoGaDB implements operator-at-a-time bulk processing model with operator-based scheduling that distributes the set of queries on all available processing resources[8] and make the most efficient use of the memory hierarchy.

Buffer management
Data placement strategy is required by the processors with dedicated memory to move data to their memory whenever needed for performing tasks and operations. CoGaDB handles this requirement using the dedicated GPU buffer manager module. GPU buffer manager[8] is responsible for providing required data to the GPU operators whenever input columns are requested by them.

Query Placement and Optimisation
GPU-accelerated database systems need to handle the environment of multiple-processing devices. CoGaGB addresses this problem of query placement and optimization by building their custom solution - Hybrid Query Processing Engine(HyPE). This module optimizes physical query plan by applying several optimization algorithms and make use of Hawk engine to produce highly optimized machine code variant for target processing device.

Consistency and Transaction Processing
It is challenging to maintain consistency in the immense multi-thread environment provided by the GPU. CoGaDB does not implement any explicit consistency standards and transactions support but provides some fault tolerance mechanism to support the durability of the system which eventually leads to consistent data storage.

4.1.3.2 Non-functional properties

Performance
S. Breß and others benchmarked CoGaDB by using Star Schema Benchmark[13], which is a popular OLAP framework frequently used for performance evaluations. By running queries provided by SSBM[6], CoGaDB GPU-acceleration can be compared and evaluated with the other versions such as MonetDB[7]. With the help of these benchmarking, performance insights of CoGaDB can be drawn as shown in Figure 4.

Portability
CoGaDB tries to achieve the hardware-oblivious design by implementing all vendor-specific operators in the system. This strategy leads to high development and implementation cost

[6]Star Schema Benchmark
[7]Open-source column-oriented database management system

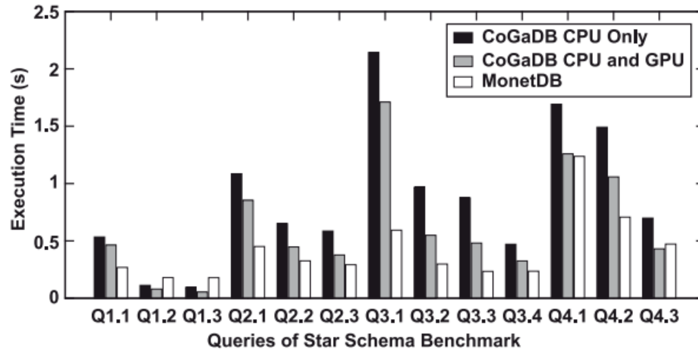

Figure 4: Response times of CoGaDB for the SSBM, taken from [8]

but provides good probability standard. Though, it still relies entirely upon the vendor-specific implementations and fails on the systems where vendor-specific GPU hardware is not present. Hence, CoGaDB doesn't have true hardware-oblivious nature.

4.2 GPUDB

4.2.1 Overview

Yuan and others started research on utilizing the power of GPU to process data warehousing queries. They believed that the implementation of such a database system with the acceleration of the GPU could provide extremely high-throughput as compared to the CPU or other processing engines. They build several modules using the drivers and libraries provided by CUDA[8] and OpenCL[9] to implement a database system called GPUDB[14], which enabled them to execute OLAP workload driven queries on the GPU.

4.2.2 Architecture

The architecture of GPUDB is shown in Figure 5. As the most DBMSs, GPUDB follows the same approach of providing SQL client interface for the user to submit queries. This input query is then passed to SQL parser which converts the provided query into the logical query plan. It is then followed by the query optimizer and query execution engine of GPUDB for generating machine code variant that can be executed on the GPU kernels. The engine also relies on the late materialization strategy[14], which fetches the required columns by the operators into the memory at the very end of the processing. Furthermore, some of the code is executed in the CPU kernels which handles the data transfer between CPU's host memory to GPU's memory and initiating GPU kernels whenever required.

4.2.3 Exploring the design-space

4.2.3.1 Functional properties

Storage system
Yuan and others have identified the solution to the PCIe transfer bottleneck limitation by introducing a crucial optimization concept of pinned host memory, which is a memory that cannot be swapped out leading to the allowance of direct access by the GPUs. Initially, GPUDB loads the entire database into CPU's host memory in order to avoid disk transfer

[8]Parallel computing platform and programming model invented by NVIDIA
[9]Framework for writing programs that execute across heterogeneous platforms such as GPUs

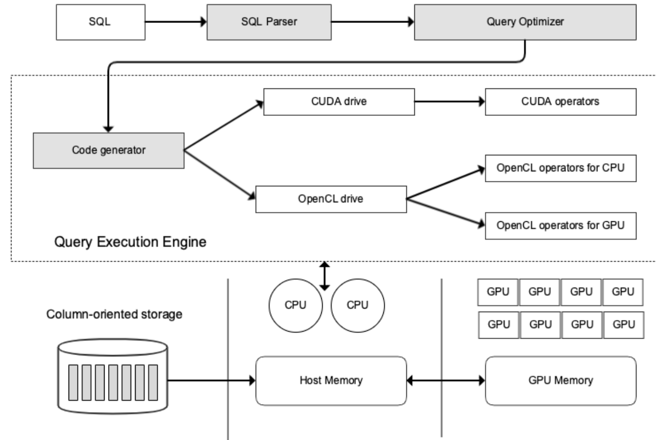

Figure 5: The architecture of GPUDB, adapted from [14]

bottlenecks and stores some of the data in pinned host memory depending upon the operators and query plan, which is directly accessible by the GPU.

Storage model

GPUDB adopted column-oriented storage as their primary storage model due to its extreme efficiency in supporting compression algorithms and materialization strategy. GPUDB's query engine supports different data compression techniques which are very effective and easy to implement in columnar storage. Thus, GPUDB directly works with compressed data for performing computations and operations whenever possible, which is supported by using the column-oriented storage model.

Processing model

As shown in Figure 5, GPUDB provides SQL client interface and interactive component for users to push their SQL queries to the database engine. For processing these queries, GPUDB uses a block-oriented processing model[15], which is a technique in which operators are processed in blocks as compared to singleton operators processing.

Buffer management

Unlike CoGaDB, GPUDB does not provide any dedicated buffer management module for implementing efficient data placement strategies. It handles its data requirements by running an isolated program interface on the host CPU's main memory which fetches data from column-oriented storage to provide it to GPU memory in late materialization manner. This program also handles data placement in the host's pinned memory to enable direct access by the GPU.

Query Placement and Optimisation

GPUDB does not implement any processor selection and cost estimation algorithm to decide query placements depending upon the requirement. It only supports the execution of any database related operations in the GPU environment using CUDA or OpenCL driver programs which are designed and optimized with pre-implemented GPU operators. GPUDB supports optimizations using its query optimizer module which induces certain performance-improving changes in the logical query plan using provided algorithms and strategies.

Consistency and Transaction Processing
GPUDB does not implement any algorithms or strategies for maintaining consistent standards for the system. After reviewing and reading several research papers, it can be concluded that GPUDB does not provide any fault tolerance policies or locking mechanism for handling transactions or inconsistency in the system. There is no module with the implementation of consistency and transaction processing in the GPUDB ecosystem.

4.2.3.2 Non-functional properties

Performance
Yuan and others benchmarked and analyzed the performance of GPUDB by using Star Schema Benchmarking (SSBM) queries to simulate OLAP workloads. They also performed an analysis of using pageable and pinned host memory for understanding the complications and limitations of PCI express bus data transfer. These performance insights can be visualized in Figure 6.

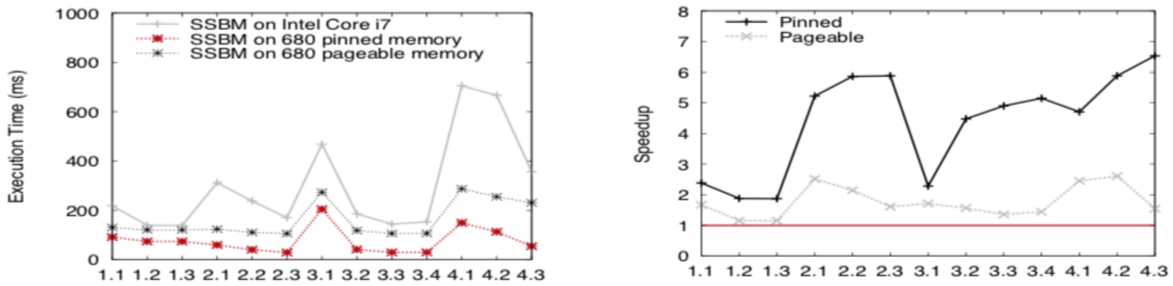

Figure 6: SSBM performance comparison, taken from [14]

Portability
GPUDB is extremely dependent on NVIDIA's CUDA libraries and OpenCL framework support providing graphics card. From this perspective, GPUDB fulfils minimal hardware-oblivious requirements as it did not implement all vendor-specific operators which are currently available in several heterogeneous processing environments. In terms of portability perspective, GPUDB needs to incorporate lots of module and vendor-specific changes in order to reach a basic design of being hardware-oblivious.

4.3 OmniSci/MapD

4.3.1 Overview

Massively Parallel Database (MapD) originated from research at MIT's Computer Science and AI Laboratory by Todd Mostak. It was developed as a data processing and visualization engine, combining traditional query processing capabilities of DBMSs with advanced analytics and visualization functionality [16].In 2017, MapD open-sourced the Core SQL Engine, built to harness the supercomputing power of GPUs.[17]. And in 2018, MapD was rebranded to OmniSci[17] Sample interactive data visualizations can be found at `https://www.omnisci.com/demos/`.

4.3.2 Architecture

MapD uses in-memory storage and leverages SSDs for persistent storage. It also uses just-in-time LLVM(Low-level Virtual Machine)-based compiler to compile SQL queries into machine

code.[16] MapD architecture implements a pyramid model, where each successive level of memory is slower computationally but larger in size than the last.[16]

OmniSci was architected for the GPU with focused development to enable common SQL analytic operations such as filtering (WHERE), segmenting (GROUP BY) and joining (JOIN) to run as fast as possible with native GPU speed.

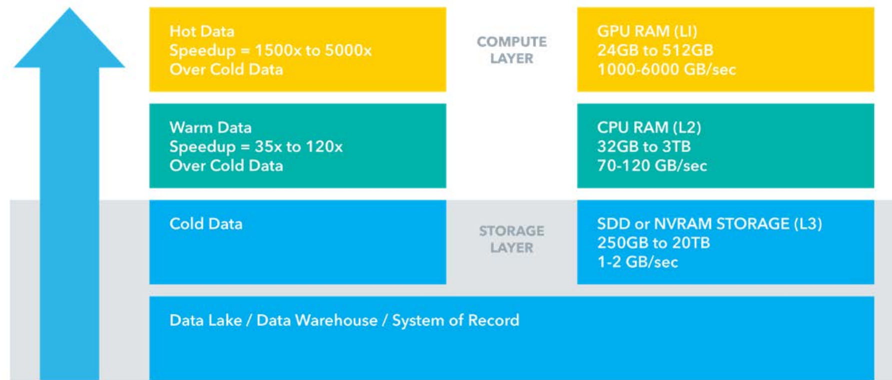

Figure 7: OmniSci/MapD architecture[18]

4.3.3 Exploring the design-space

4.3.3.1 Functional properties

Storage system
OmniSci/MapD GPU-powered SQL data platform uses in-memory storage and leverages modern SSDs for persistent storage. MapD tries to avoid disk access by keeping most of its data in memory. In contrast to other GPU systems that store data in CPU to transfer it during query time, OmniSci Core caching up to 512GB of the most recently touched data to avoids this transfer inefficiency[18].

Storage model
OmniSci/MapD stores data in a in the column format and partitions columns into chunks. Columns that are frequently used for filters are cached in the GPU memory.[16]

Processing model
The basic processing model of MapD is processing one operator-at-a-time. A chunk is the basic unit of MapD's memory manager. Due to the partitioning of data into chunks, it is possible to process data on a per chunk basis. Hence, MapD is capable of applying block-oriented processing[16].

Buffer management
When queries are executed, the OmniSci Core database optimizer utilizes GPU RAM first if it is available. You can view GPU RAM as an L1 cache conceptually similar to modern CPU architectures. The OmniSci Core database attempts to cache the hot data. If GPU RAM is unavailable or filled, the OmniSci Core database optimizer utilizes CPU RAM (L2). If both L1 and L2 are filled, query records overflow to disk (L3).[19]

Query Placement and Optimisation
MapD applies a streaming mechanism for data processing. The optimizer tries to split a query plan in parts, and process each part on the most suitable processing device.[16] Within one node, OmniSci uses a shared-nothing architecture between GPUs. On query processing each GPU processes independently processes data, from other GPUs.[18]

OmniSci uses a JIT (Just-In-Time) compilation framework built on LLVM (Low-level Virtual Machine) that allows OmniSci to transform query plans into an architecture independent intermediate representation code (LLVM IR)[18]. OmniSci also improves performance using vectorization[18].

Consistency and Transaction Processing
OmniSci/MapD ensure consistency in a distributed system with method of 'Immediate/Strong consistency'.

4.3.3.2 Non-functional properties

Performance
Compared to traditional data warehouses that used techniques such as indexing, downsampling or pre-aggregation for an increase of performance, OmniSci Core leverages GPUs to achieve the massive performance gains[18]. Coupled with its execution engine, GPUs deliver instantaneous query execution without the need to index, downsample or pre-aggregate beforehand[18]. This approach has two major upsides. First, organizations deploying OmniSci can simply load the data and use fast SQL without spending significant time and resources modeling[18]. Second, OmniSci can load the data quickly due to the lack of any significant preprocessing.[18]

Portability
OmniSci has taken GPU-native approach. Its fast hardware requires software designed specifically for the GPU's unique advantages. As a result, OmniSci's uses software to harnesses the hardware attributes that deliver analytic performance at speed, even when running on hardware with both GPUs and CPUs.[18]

4.4 Brytlyt

4.4.1 Overview

Brytlyt combines GPU hardware for processing joins in parallel(Using patent-pending intellectual property). Brytlyt is built on PostgreSQL and supports all its features, including stored procedures, full JSON support, and PostgreSQL's native data connectors. It can easily leverage existing technology as well as connect directly to any existing data source. Hence it is easy to integrate Brytlyt with existing investments, lowering time-to-value, and increase ROI[20]. Brytlyt's solution drastically improves data processing power without the corresponding significant financial investment.

There are multiple factors involved in the selection of a GPU accelerated database. Figure 8 will make it clear.

4.4.2 Architecture

Brytlyt supports JOIN operations directly on the GPU. Data is broken in blocks and distributed to GPU cores that search through using horizontal partitioning.[22]

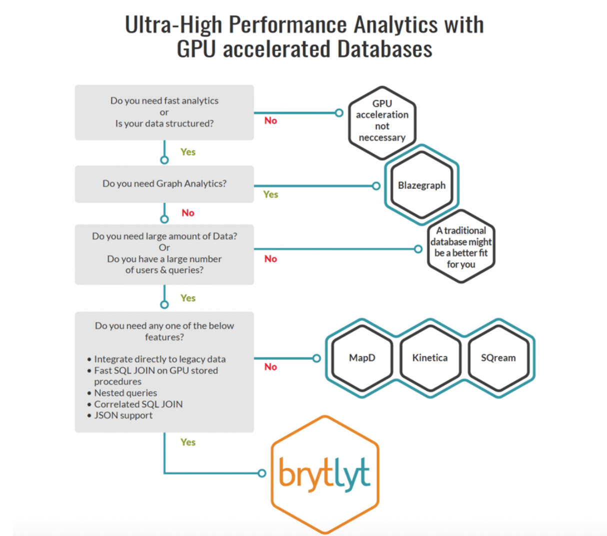

Figure 8: Factors for GPU Accelerated Database Selection[21]

Brytlyt' patent-pending method recursively searches and separates the rows into a 'hit' and 'not hit'. These operations run in parallel[20]. Blocks of broken data are distributed to the parallel cores for searching. For example, a dataset of 400,000 rows can be broken in 200 rows for 2000 GPU cores running in parallel and working independently on their blocks of data. The process runs recursively until only the relevant block remain and empty ones are discarded.[20]

Brytlyt supports basic SQL operations such as filters, sorts, aggregation, groups, and joins[22]. Data is stored in vectorized columns and transferred between CPU and GPU memory during an execution cycle[22].

4.4.3 Exploring the design-space

4.4.3.1 Functional properties

Storage system

Brytlyt is a disk-oriented database. Data is stored on disk using the table and index. Data transfer is fully scheduled between GPU and CPU memory to reduce the data transfer overhead.[22]

Storage model

Brytlyt database uses the storage model that is derived from PostgreSQL, which stores data using N-ary storage model[10].

Processing model

Brytlyt has adopted the open source database PostgreSQL on which it implements its intellectual property on SQL operations. With PostgreSQL, brytlyt can extend the full suite of SQL and programmatic SQL functionalities.

In Brytlyt, queries are executed on both CPU and GPU. For CPU Tuple-at-a-Time model is much more suitable, however, for the GPU acceleration the vectorized method to increase the parallelism is used. Vectorized data processing speeds utilizes the speed of modern hardwares for building highly efficient analytical query engines[20].

[10] https://www.postgresql.org/docs/9.4/storage.html

Buffer management
Brytlyt mostly keeps its data in the main system memory to improve the performance, since the performance of GPU-accelerated database highly depends on the data PCIe transfer efficiency. And the data transfers can be significantly accelerated by keeping 'semi-hot data' in host memory and hot data in GPU RAM. In order to avoid PCIe bottlenecks Brytlyt maintains a 1:1 ratio of CPU and GPU to utilize the full capabilities of the system.[20]

Query Placement and Optimisation
Main difference between CPU and GPU in-memory database is how they store and process data. Data usually resides in CPU memory in vectorized columns to optimize parallel processing across all available GPUs. The data is moved as needed to GPU memory for both mathematical and spatial calculations, and the results then returned to CPU[20]. For smaller data sets and live streams, the data can reside entirely in the GPU's for even faster processing.[20]

Consistency and Transaction Processing
Brytlyt follows 'Immediate/Strong consistency'. In a lecture presented by Richard Heyns[23], the CEO of Brytlyt, they support PostgreSQL's WAL mechanism [11] and have no logical changes in concurrency control levels, which are derived from PostgreSQL

4.4.3.2 Non-functional properties

Performance
Brytlyt gained huge performance improvement over traditional databases due to its architecture and specialized execution of SQL operations. It could perform around 300 times faster during independent benchmark test.[12][20] Brytlyt performance results(figure 9) were gained from indicative testing based on the TPC-H Query 1 out of multiple queries.[20]

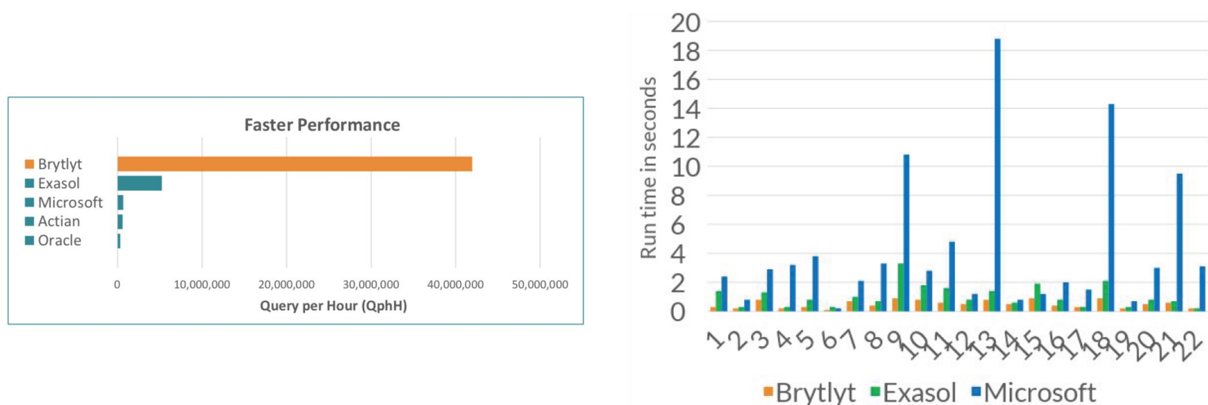

Figure 9: Performance comparison of Brytlyt[20]

Portability
One of the major advantage of Brytlyt is that is it built on the PostgreSQL and is complimented with its deep functionality of SQL on PostgreSQL. Current investments in code, analytics and visualisation remains untouched and above that processing can be accelerated with integration of Brytlyt with little to no effort. Also integration is almost seamless with existing systems using data connectors.[24]

[11]https://www.postgresql.org/docs/9.1/wal-intro.html
[12]https://tech.marksblogg.com/benchmarks.html

4.5 SQream

4.5.1 Overview

SQream[13], found in Isreal by Ami Gal and Kostya Varakin, provides a GPU accelerated data warehouse for data scientists, business intelligence professionals and developers to execute complex SQL queries to gain insights from large dataset using the tools they use today. SQream DB exploits GPU computing power to increase the performance of columnar queries by at least 20 times on large dataset.

4.5.2 Architecture

While traditional database should increase nodes of CPU to running massively parallel operations, SQream DB uses both CPU and GPU to process large dataset, and especially running massively parallel computing on GPU since GPU perform very well with repetitive operations on large amounts of data parallelly. For those operations, which are computing different logics and not easy to be paralleled, are best performed on the CPU. Therefore, the compiler will decide which operations should be performed on CPU and the others be performed on GPU. As a result, SQream DB is able to make the most from both CPU and GPU.

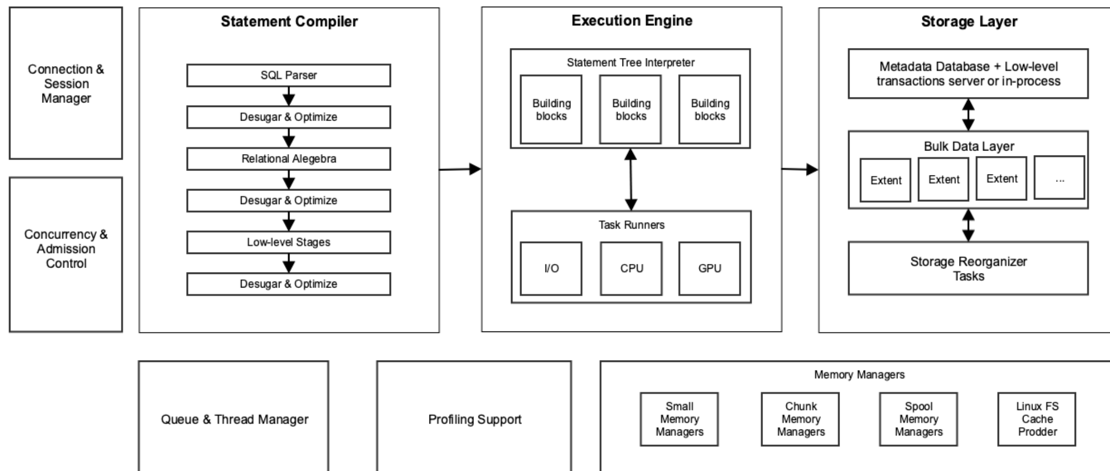

Figure 10: SQream DB architecture [25]

4.5.3 Exploring the design-space

4.5.3.1 Functional properties

Storage system

SQream DB persists data on the disk but reads the necessary data into main memory for further operations[26]. However, fetching performance becomes the biggest issue in a query's runtime. This meant it would also be the biggest bottleneck. SQream uses the GPU compressing and filtering technique to reduce I/O bottlenecks.

Storage model

SQream DB is a columnar database designed for GPU-accelerated computing. We consider the columnar approach as a vertical partition which operate very well on GPU. SQream DB

[13]https://en.wikipedia.org/wiki/SQream_DB/

also provides horizontal partitioning which splits up data into chunks. These chunks then are compressed by the algorithms and become small, which makes them efficient for transfer across the PCI bus to the GPU for processing.

Processing model
The processing model of SQream DB is operator-at-a-time. It could also apply block-oriented processing since it is capable to process by a chunk. SQream DB provide interface for user either directly querying by SQL-92 standard or through a connector like ODBC or JDBC, as well as native Python and ADO.Net[27].

Buffer management
SQream DB is capable to compresses data all the time and that minimizes the I/O to about 20-25 percent to communicate to the data. In addition to accessing for query speed, It can ingest data at approximately 3 terabytes per hour per GPU so it has an initial load and then it can continually insert new data and delete old data. Because of the data have been compressed and break into chunks, the data can easily place that on the flash storage.

Query Placement and Optimisation
SQream DB has query compiler and plan optimizer to decide the operations should perform on GPU or on CPU. For instance, since GPU is good at computing repetitive and parallel process but not good for performing text operations, the compiler will make these happen on CPU. In the converse, the compiler will decide which columns will go up to the GPU for processing, if the tasks can be parallelized.

Consistency and Transaction Processing
SQream DB is capable to support large amount of users due to its unique shared-data architecture. It also comply with atomicity, consistency, isolation and durability(ACID), which means it has transparent commit, rollback and recovery, with full isolation. That is to say, the data is always safe and consistent in SQream DB.

4.5.3.2 Non-functional properties

Performance
SQream claims that SQream DB could process up to 100 times faster than other relational databases with a large dataset. SQream provides an example which doing 85 terabytes using Apache Phoenix, a relational database engine, and it takes five hours to get through, while SQream DB takes the queries down to five minutes. Therefore, SQream DB outperforms than traditional database which running only on CPU with a large dataset.

Portability
SQream DB is tailored to NVIDIA's CUDA libraries, therefore it runs on Nvidia Tesla GPU which is coupled to whatever CPU customers can pair it with, typically a multicore Intel Xeon. Recently, SQream DB releases its version 3, changing from Xeon processors to IBM's Power9 processors' pair with Nvidia Tesla V100 GPU. As a result, SQream DB is not a hardware-oblivious database.

4.6 PGstrom

4.6.1 Overview

PG-Strom is an extension software for PostgreSQL(v9.5 or later) to speed up complex SQL workloads with a large dataset. PG-Strom uses GPU to scan a large number of records with complex qualifiers[28]. PG-Strom compiles relative GPU code rapidly according to SQL optimizer decisions of the custom plans which use GPU for SQL execution.

4.6.2 Architecture

As Figure 11 shown, we usually need to load whole data block on the storage to CPU even though there are junk data for CPU, but PG-Strom presents SSD-to-GPU Direct SQL Execution, which loads the entire data block to GPU RAM by GPUDirect RDMA. At first, SQL parser breaks down the supplied SQL queries to parse tree. Then GPU code generator that automatically generates GPU program according to the SQL commands and asynchronous parallel execution engine to run SQL workloads on GPU device[29]. These codes then provide to execution engine to pre-process data on GPU device to reduce data size prior to its arrival at CPU/RAM.

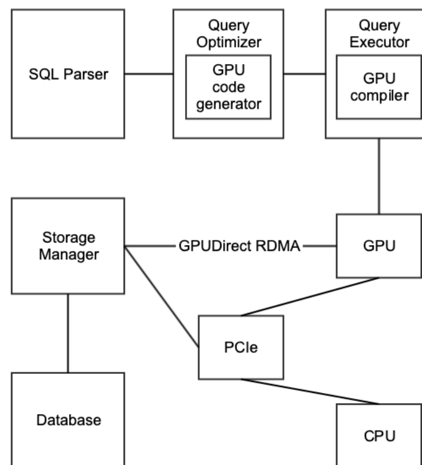

Figure 11: PG-Strom architecture adapted from [30]

4.6.3 Exploring the design-space

4.6.3.1 Functional properties

Storage system
Because PG-Strom utilizes PostgreSQL as its database system, we can say that it is a hard-disk-based system. However, it's different from other DBMSs designed on typical HDD. Instead, it exploits the performance benefits from using SSD. Although the PCIe transfer is expensive and is considered as the major overhead in GPU execution, PG-Strom has the unique feature, SSD-to-GPU Direct SQL Execution, to bypass the PCIe transfer. This bypass allows to pull out nearly wired performance of the hardware[30].

Storage model
PG-Strom is a row-oriented storage model because of utilizing PostgreSQL. However the latest PG-Strom version supports to transfer data structure from row-oriented to column-

oriented on GPU device memory. Using columnar format significantly reduces the amount of data, this results in efficient utilization of bandwidth throughout the storage hierarchy.

Processing model

PG-Strom enhance query processing and performance by utilizing all available processors according to operators. It implements a batch-oriented, operator-based query execution scheme. Under this scheme, table data are splits up to large chunks and pushed from one operator to another for processing.

Buffer management

PG-Strom uses PostgreSQL heap buffer for providing required data to the GPU operators whenever the data are requested[30]. This buffer controls the data transfer between shared memory and persistent storage, which works very efficiently. PG-Strom also supports the demand paging of GPU device. Thus it only loads pages that are demanded by the executing process[30].

Query Placement and Optimisation

The query optimizer of PG-Strom would collaborate with query execution planner of PostgreSQL. If the estimated cost for those execution plans on CPU is higher, it offers alternative query execution plans for GPU. However, it requires operators, functions and data types in use must be supported by PG-Strom for GPU execution[28].

Consistency and Transaction Processing

PG-Strom relies on PostgreSQL transaction strategy, it thus supports implementation of concurrency and full respect of the system's constraints and properties when multiple transactions are modifying the state of the system at the same time[31]. The concurrency issue can be dealt with gracefully because PostgreSQL is fully ACID(atomicity, consistency, isolation, and durability) compliant and implements transactions isolation.

4.6.3.2 Non-functional properties

Performance

PG-Strom with SSD-to-GPU Direct SQL Execution could speed up average 3.5 times faster[30]. PG-Strom allows the data load to GPU directly and bypasses the PCI express, which cost expensively on transferring data between CPU and GPU. With this advantage, we could exploit the parallel computing power from GPU.

Portability

According to the supplied SQL, PG-Strom internally generated GPU programs by CUDA language. CUDA is a programming environment provided by NVIDIA, which allows implementing the parallel program on GPU device. That is to say, PG-Strom is not a hardware-oblivious database.

4.7 OmniDB

4.7.1 Overview

OmniDB[14] is an interactive and powerful, yet lightweight, browser-based database management tool and it is developed by Zhang, He and others[15]. OmniDB provides a unified workspace interface which allows users to manage diverse databases. In addition, it supports diverse database systems, such as Oracle databases, MySQL and MariaDB. Unlike, C-Store, GPUQP and StagedDB, OmniDB can perform in any browser and operating system[32].

4.7.2 Architecture

In OmniDB architecture, portability and efficiency are regarded as two main keys. To maintain both key properties of OmniDB architecture, a kernel-adapter based implementation approach is required. As shown in Figure 12, the architecture is composed of a query processing kernel, known as qKernel, and architecture-aware adapters.

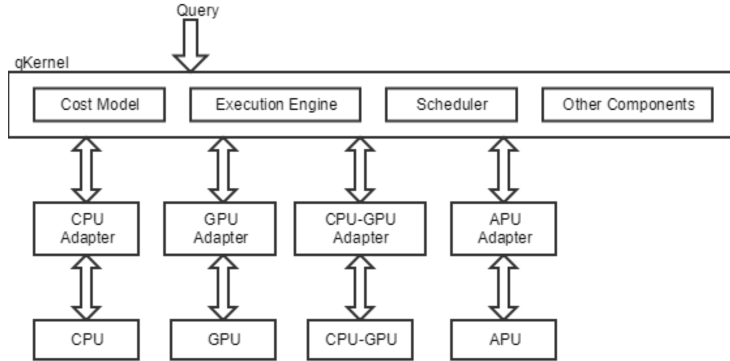

Figure 12: OmniDB - Kernel Adapter Desgin from [32]

The qKernel includes an execution engine, a scheduler, a cost model and so on. To be specific, the execution engine helps query processing operators to process the data in parallel. In addition, the scheduler assigns work units to individual PPEs[16]. The work unit represents a certain amount of workload for each PPE's per scheduling. Also, the cost model is used for estimating the total cost of executing a work unit. Additionally, each adapter consists of the software components, parameters and configurations to adapt qKernel to the target architecture[32].

4.7.3 Exploring the design-space

4.7.3.1 Functional properties

Storage system
OmniDB storage system uses CPU main-memory as its primary data storage location. It is based on the conclusive fact that in-memory processing is faster than disk based processing. Also, it saves data in GPU memory cache to improve performance and efficiency.

[14]https://www.2ndquadrant.com/en/resources/omnidb/
[15]https://code.google.com/archive/p/omnidb-paralleldbonapu/
[16]Parallel-Processing Elements

Storage model
OmniDB storage model is structured to fit the hardware capabilities of modern CPUs and GPUs. Hence, OmniDB uses columnar storage data model for highly efficient query processing.

Processing model
OmniDB processing model manages both scheduling and analyzing work units. Individual work units can have different segments such as an operators, query or chunks of tuples. In order to process these work units, OmniDB implements block-oriented processing model as it is highly efficient for working with several chunks of tuples[32].

Buffer management
OmniDB adapts parallel CPU/GPU architectures which consists of multiple threaded-parallel processing elements. The transfer requirement depends upon the location of element execution. It can be stated that OmniDB uses several custom constructs to fulfill data transfer requirements from CPU to GPU, unlike CoGaDB which provides seperate dedicated module.

Query Placement and Optimisation
OmniDB utilises the highest throughput processing device for each work unit which helps in providing extremely efficient query placement. OmniDB has a scheduler which helps to make sure the workload does not exceed a certain amount of average workload on the target processing device. In addition, since OmniDB uses kernel-based approach, its cost model is deeply related to the adapters which helps in implementing certain optimisation standards.

Consistency and Transaction Processing
OmniDB does not implements any consistency standards in the database system. There is no transaction support, recovery mechanism or fault tolerance designed due to the complications introduced by GPU architecture and parallel job executions.

4.7.3.2 Non-functional properties

Performance
When Zhang and the others analyzed OmniDB, they implemented it to four different kinds of targets which are CPU-only, GPU-only, CPU-GPU and APU. The performance of OmniDB can be divided into three parts. Firstly, the effectiveness of adapters on each architecture is evaluated. The result of this one is that the moderate size of work unit is the most efficient and maximises the performance of query processing. Secondly, how different work units impacts on the scheduling algorithm of the homogeneous architectures, such as CPU-only and GPU-only. The utilised scheduling algorithm represented the higher throughput performance compared to the FIFO scheduling algorithm with 8 to 33% of the improvement result on the CPU-GPU and 4 to 19% on the APU architectures. Lastly, they analysed the profiler result of OmniDB. As an example, they compared the normal hash joins on the NVIDIA GPU profiler with CPU systems and are able to get 20% improvement[32].

Portability
OmniDB can provide hardware-oblivious design due to its implementation of different adapters for a specific set of operators and cost functions[32]. This database system is not designed with vendor-specific frameworks and programming constructs due to which it can provide highly efficient portability.

4.8 Virginian

4.8.1 Overview

Virginian was developed by Bakkum and others to utilises the power of GPU to perform heavy computational query processing with efficient caching. It also provides a service for both CPU and GPU execution of queries[33]. It is designed as an experiment at NEC laboratory in America to test and compare data processing on the CPU with GPU specifically NVIDIA.

4.8.2 Architecture

One of the most significant features of Virginian architecture is memory uniting. Each thread in a thread block approaches the global memory of GPU concurrently with proper arrangement. Thus, the GPU hardware coalesces them in one memory fetch. Bakkum and others also built a data structure, named the Tablet as shown in Figure 13, and the tablet helps the GPU to manage information flexibly. Since the data structure has a high tolerance of execution speed and the relative speeds of CPU compared to GPU when they are executed, this data structure is utilised for query execution. It can be used efficiently for both CPU and GPU execution. Other than those features, there are diverse characteristics to manage the data structure. Therefore, the speed of GPU query execution increases [33].

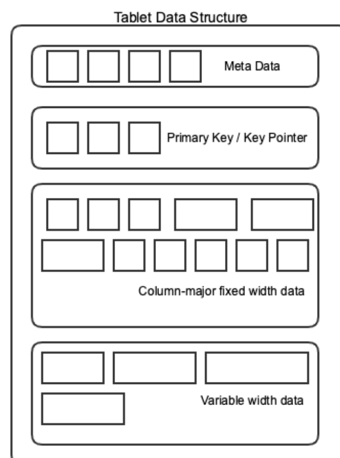

Figure 13: Tablet Data Structure from [33]

4.8.3 Exploring the design-space

4.8.3.1 Functional properties

Storage system
Virginian utilises uniform virtual addressing (UVA) for its storage system, so it does not adapt convention caching of operators. By using the uniform virtual addressing, a GPU kernel is able to connect data stored right away. After that, the accessed data is delivered to the device via the bus directly and efficiently.

Storage model
As it is presented above, Virginian utilises Tablet as a data structure. This data structure saves certain size of data in column-oriented layout. In addition, the Tablet is also able to store variable sizes of data like strings or VARCHAR.

Processing model
As a basic query processing model, Virginian adapts operator-at-a-time processing. Also, it utilises an alternative processing scheme. Unlike the other systems, Virginian uses an opcode model for query execution. Using the opcode allows Virginian to avert block-wise processing on the GPU which is a result of writing and reading redundantly.

Buffer management
Virginian adapts opcode to manage their memory efficiently. Query is broken down into several steps of opcodes and data transfer requirements for these opcodes is fulfilled by in-built custom program.

Query Placement and Optimisation
Generally, query processing is able to implement either on the CPU or on the GPU in Virginian. Therefore, the workload in Virginian does not need to be divided between CPU and GPU. Also, unlike other DBMSs such as OmniDB and MapD, query optimiser which works for both does not exist in Virginian.

Consistency and Transaction Processing
Virginian provides reduced memory transaction. This is because by using the same kernel operation, it does not require the movement of intermediate state [33]. Other than this, there is no construct of consistency management or transaction processing provided in any of the research papers related to Virginian.

4.8.3.2 Non-functional properties

Performance
Bakkum and others implemented the dataset composed of about eight million rows of arbitrarily generated numerical values. They also executed five different categorised configurations which are single core, multi-core, serial, mapped and cached. According to Figure 14, it shows that mapped and cached is much efficient compared to serial. It also shows that single core execution took the longest time compared to the rest of them and the average execution time of the mapped GPU and the cached GPU are also the lowest[33].

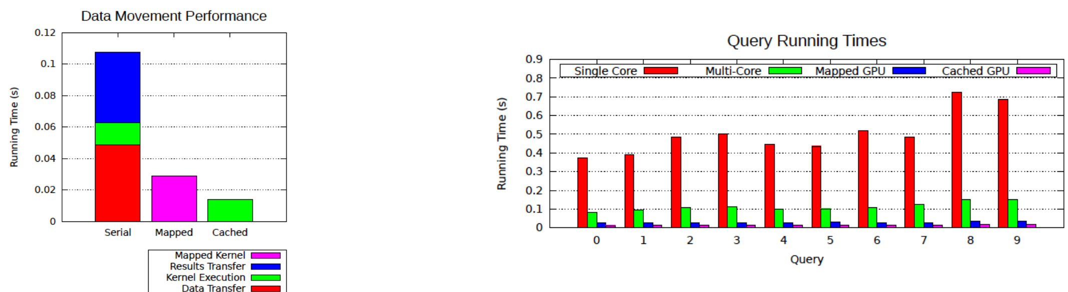

Figure 14: Data Movement and Query Runtime Performance, taken from [33]

Portability
Virginian works by implementing certain NVIDIA specific operators in their opcodes generation since it is designed to compare performance of CPU with NVIDIA GPUs. Hence, Virginian does not support hardware-oblivious design like most of other GPU-accelerated database systems stated above.

5 GPU-accelerated Database Systems Comparison

In this section, we are comparing above surveyed GPU-accelerated database systems with respect to their design-space. Using this comparison, we can relate how different database systems work in the GPU-environment which will help us in building a common reference design for such systems.

5.1 Functional Properties

5.1.1 Storage System

All 8 GPU-accelerated database systems reviewed do not have generalized storage system implementations. As shown in the below comparison table, CoGaDB and GPUDB uses main-memory as their underlying storage system but do not provide any support for SSD integration and GPU caching. In comparison, MapD provides caching support of up to `512GB` in GPU's device memory with seamless SSD integration. On the other hand, Brytlyt and PGstrom use PostgreSQL as their database engine due to which they are more dependent on disk-based storage system where PGstrom supports SSD with SSD-to-GPU direct SQL execution feature. Finally, SQream is more driven on supporting hundreds of terabytes of data whereas OmniDB and Virginian are based upon the early implementation of these systems which makes them lacking most of the new features such as SSD integration support.

GPU-DBMS	**Storage System**		**Storage Features**	
	Disk-based	**Main-memory**	**GPU Caching**	**SSD Support**
CoGaDB	××	✓✓	××	××
GPUDB	××	✓✓	××	××
OmniSci/MapD	✓✓	✓✓	✓✓	✓✓
Brytlyt	✓✓	××	××	××
SQream	✓✓	××	××	××
PGstrom	✓✓	××	××	✓✓
OmniDB	××	✓✓	××	××
Virginian	××	✓✓	××	××

Table 1: Comparison of Storage System (✓ - Supported, × - Not Supported)

5.1.2 Storage Model

After surveying these database systems, it can be concluded that GPU-acceleration performance and implementation is strongly dependent on using column-oriented storage as their underlying storage model. All 8 databases systems as shown in below comparison table supports column-store where Brytlyt and PGstorm also support row-store due to PostgreSQL being their underlying database engine. Moreover, GPUDB supports additional features such materialization strategy and several compression techniques which can only be implemented in a column-oriented data structure. MapD and SQream support partitioning as their additional feature, where SQream provides both horizontal and vertical partitioning of the data. Using this feature, data is split into small chunks which increases transfer throughput and processing speed for queries in the GPU environment.

GPU-DBMS	Storage Model		Model Features	
	Row Store	Column Store	Compression	Partitioning
CoGaDB	××	✓✓	××	××
GPUDB	××	✓✓	✓✓	××
OmniSci/MapD	××	✓✓	××	✓✓
Brytlyt	✓✓	✓✓	××	××
SQream	××	✓✓	××	✓✓
PGstrom	✓✓	✓✓	××	××
OmniDB	××	✓✓	××	××
Virginian	××	✓✓	××	××

Table 2: Comparison of Storage Model (✓ - Supported, × - Not Supported)

5.1.3 Processing Model

In these 8 GPU database systems, there are no database systems using tuple-at-a-time as their processing model. Unlike the traditional database system, operator-at-a-time or block-at-a-time processing model allows GPU to exploit its parallel performance. Except for CoGaDB and Brytlyt, other systems support both operator-at-a-time or block-at-a-time.

GPU-DBMS	Processing Model		
	Tuple-at-a-Time	Operator-at-a-Time	Block-at-a-Time
CoGaDB	××	✓✓	××
GPUDB	××	✓✓	✓✓
OmniSci/MapD	××	✓✓	✓✓
Brytlyt	××	✓✓	××
SQream	××	✓✓	✓✓
PGstrom	××	✓✓	✓✓
OmniDB	××	✓✓	✓✓
Virginian	××	✓✓	✓✓

Table 3: Comparison of Processing Model (✓ - Supported, × - Not Supported)

5.1.4 Buffer Management

All of the 8 database systems surveyed implements data placement strategies to move data from CPU to GPU in the most optimized manner. All of these strategies are specific and customized with respect to the system and its architectural design. CoGaDB, MapD, and PGstrom implement a dedicated module for supporting the most efficient buffer management specification. On the other hand, GPUDB, Brytlyt, OmniDB, and Virginian provides custom functionalities to transfer data from CPU to GPU effectively.

5.1.5 Query Placement and Optimisation

As per the survey, we found that six of databases: CoGaGB, PGstrom, SQream, OmniSci, Brytlyt, and OmniDB, use some form of the hybrid query optimizer. They implement optimiser to split query plans into parts and processes each part on the most suitable processing device. While two other databases: GPUDB and Virginian, don't have any specific hybrid query optimizer, they use query optimiser modules on query plans while transforming queries into CUDA or OpenCL driver programs.

GPU-DBMS	Buffer Management Module		
	Dedicated	Custom	Data Placement Strategies
CoGaDB	✓✓	××	✓✓
GPUDB	××	✓✓	✓✓
OmniSci/MapD	✓✓	××	✓✓
Brytlyt	××	✓✓	✓✓
SQream	××	✓✓	✓✓
PGstrom	✓✓	××	✓✓
OmniDB	××	✓✓	✓✓
Virginian	××	✓✓	✓✓

Table 4: Comparison of Buffer Management (✓ - Provided, × - Not Provided)

GPU-DBMS	Query Placement and Optimisation	
	Hybrid query optimiser	Non-hybrid query optimiser
CoGaDB	✓✓	××
GPUDB	××	✓✓
OmniSci/MapD	✓✓	××
Brytlyt	✓✓	××
SQream	✓✓	××
PGstrom	✓✓	××
OmniDB	✓✓	××
Virginian	××	✓✓

Table 5: Comparison of Query Placement and Optimisation (✓ - Supported, × - Not Supported)

5.1.6 Consistency and Transaction Processing

Based on the survey we know that CoGaDB, GPUDB, OmniSci, OmniDB, and Brytlyt does not implement any explicit consistency standards. PGstrom and SQream manage consistency by implementing ACID(atomicity, consistency, isolation and durability) properties. Overall, CoGaDB, OmniSci, OmniDB, Brytlyt, PGstrom and SQream maintain Immediate consistency wile GPUDB maintains Immediate/strong Consistency or Eventual Consistency depending on the configuration. No sufficient data found about Virginian.

GPU-DBMS	Consistency and Transaction Processing		
	Eventual	Immediate	ACID Properties
CoGaDB	××	✓✓	××
GPUDB	✓✓	✓✓	××
OmniSci/MapD	××	✓✓	××
Brytlyt	××	✓✓	××
SQream	××	✓✓	✓✓
PGstrom	××	✓✓	✓✓
OmniDB	××	✓✓	××
Virginian	-	-	-

Table 6: Comparison of Consistency and Transaction Processing (✓ - Supported, × - Not Supported)

5.2 Non-Functional Properties

5.2.1 Performance

After analysing the eight GDMS stated above, it can be concluded that all these systems provide better performance than CPU. Some systems such as Brytlyt and SQream provides 100-300x better performance on OLAP and OLTP workloads. Other systems also provide similar speedups which clearly concludes the immense practical utility of GPU-accelerated database systems in the computing field.

GPU-DBMS	**Performance**		
	Benchmark	**Better than CPU**	**Claimed Metrics**
CoGaDB	SSBM	✓✓	34x speedup
GPUDB	SSBM	✓✓	10x speedup
OmniSci/MapD	OLAP, OLTP	✓✓	-
Brytlyt	TPC-H	✓✓	300x speedup
SQream	85 TB	✓✓	100x speedup
PGstrom	PostgreSQL(CPU)	✓✓	3.5x speedup
OmniDB	OLAP	✓✓	27% faster joins
Virginian	OLAP	✓✓	17-35x speedup

Table 7: Comparison of Performance

5.2.2 Portability

According to the comparison of portability of the mentioned GDBMS, OmniDB is the only one being able to support hardware-oblivious database architecture.

GPU-DBMS	**Portability**	
	Hardware Aware	**Hardware Oblivious**
CoGaDB	✓✓	××
GPUDB	✓✓	××
OmniSci/MapD	✓✓	××
Brytlyt	✓✓	××
SQream	✓✓	××
PGstrom	✓✓	××
OmniDB	××	✓✓
Virginian	✓✓	××

Table 8: Comparison of Portability (✓ - Supported, × - Not Supported)

6 Open Challenges and Research Questions

In this section, we are identifying potential open challenges and research-oriented question for GPU-accelerated database systems. These challenges are the major bottlenecks and issues in the current architecture and implementation of these systems.

1. **PCIe transfer bottleneck.** One of the most challenging bottleneck and open research question which is impacting every above surveyed systems. As shown in the GPU architecture in Figure 2, PCIe bus is required to transfer data from CPU main-memory to GPU device memory for computations and analysis, which becomes extremely slow for large data. This limitation is handled by several systems using different techniques such as utilizing pinned memory (GPUDB) and creating a unique feature like SSD-to-GPU direct SQL execution (PGstorm). Now, NVIDIA has designed a new hardware fabric named NVLink[17] that can provide 10x more bandwidth than PCIe, thus breaching this limitation. Though, this state of the art hardware is still very upcoming, extremely expensive and currently only available on NVIDIA P100 GPUs.

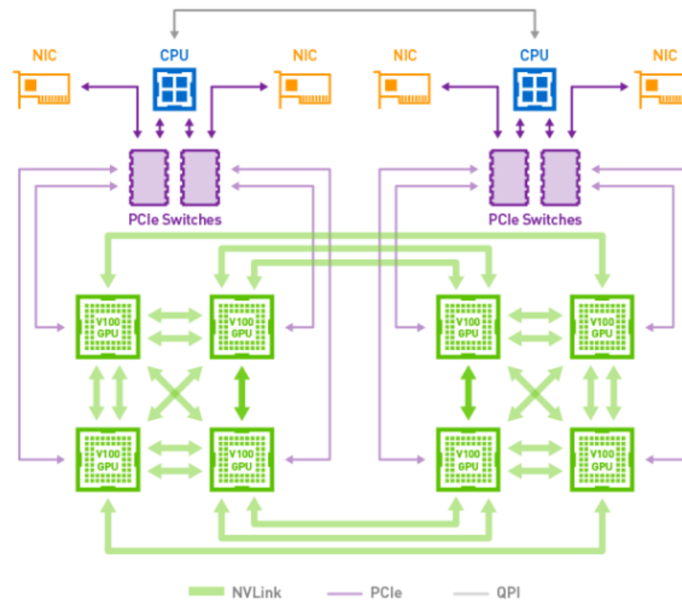

Figure 15: NVLink providing 10x more bandwidth than PCIe[34]

2. **Optimising Data Placement Requirements.** Due to the above discussed limitation and limited size of GPU device memory, understanding the right data which should be transferred to GPU is extremely challenging. Databases like CoGaDB applies several optimising and analyzing strategies to understand the data requirements of the query and try to transfer related data only. But, it is still very difficult to understand entire data requirement of the query which makes this an open research-oriented problem. If these strategies are not in place in an optimized manner, then these database systems will have major performance hits as they will require data for analysis which is still not transferred to GPU's device memory.

[17] https://www.nvidia.com/en-au/data-center/nvlink/

3. **Columnar Storage Strategy.** Since all of these GPU-accelerated databases are column-oriented databases, there are several difficulties to overcome in order to exploit the benefits from column-oriented storage. First, accessing multiple columns at one time for computing purpose is random and discontinuous, and the random access to the hard disk will seriously affect the performance. Increasing the buffer area for storing the retrieved data can reduce the time for accessing the hard disk. Second, for parallel processing, it's necessary to have data segmented. However, segmenting data within a column may not fall in the same record as a corresponding point within another column, resulting in misarranged data. Therefore, the block-based segmentation strategy is a way to deal with this problem, but the strategy has its own difficulties to cope with as well. In conclusion, in order to perform efficiently on column-oriented storage, these GPU-accelerated databases should come up with different approaches to solve these problems of columnar storage[35].

4. **Query Compilation for Multiple Devices.** In traditional databases systems, the query operation only runs on CPU which makes the query optimization straightforward. However, While some of the GPU-accelerated databases, such as SQream DB, utilize both CPU and GPU for query operations, the strategy of query optimization and query compilation makes a big impact. To exploit the biggest performance of GPU computing, the strategy has to assign those task which could be parallel processed concurrently. In the end, both CPU and GPU should cooperate well to improve the performance. Thus, the query compilation strategy is a challenge for GPU-accelerated databases.

5. **Disk-IO bottleneck.** GPUs will not improve performance for disk-based database systems, since most of the time will be spent in disk IO. GPUs improve performance only when data is in main system memory, hence it's much better to keep hot data in main memory.[20]

6. **Efficient parallel processing for SQL joins.** The traditional approach for running joins on CPU and is not well suited for the hundreds of thousands of cores in a GPU system. Since GPU's have cores grouped in chunks, with each chunk running the same instructions, most GPU Databases have a tough time with join operations. There is however one GPU Database that has solved this challenge – Brytlyt.[20]

7. **Implementation in Diverse Architecture.** Since OmniDB does not fully support on all architectures, this requires to do a future enhancement. Therefore, it needs to be evaluated on other systems including Intel Xeon Phi[32].

8. **Multi GPU and GPU/CPU concurrency.** Currently, OpenCL compilers do not support enough functions for concurrency so it causes several errors related to it. By enabling multi-GPU and GPU/CPU concurrency, the total productivity will be increased[36].

7 Conclusion

In this paper, we surveyed different GPU-accelerated database systems by exploring their design-space. Using this survey, we are able to summarise our understanding on the basis of different functional and non-functional parameters which helped us in evaluating several scenarios for identifying the most appropriate GPU-aware DBMS. In summary, each of these database systems can be used in different scenarios which is visualized as a flowchart in Figure 16. There can be other scenarios in which these systems and also other systems which we are not reviewing might be more appropriate, but this visualization can help us understand how GPU-accelerated database system fit in the data analytics and computing field.

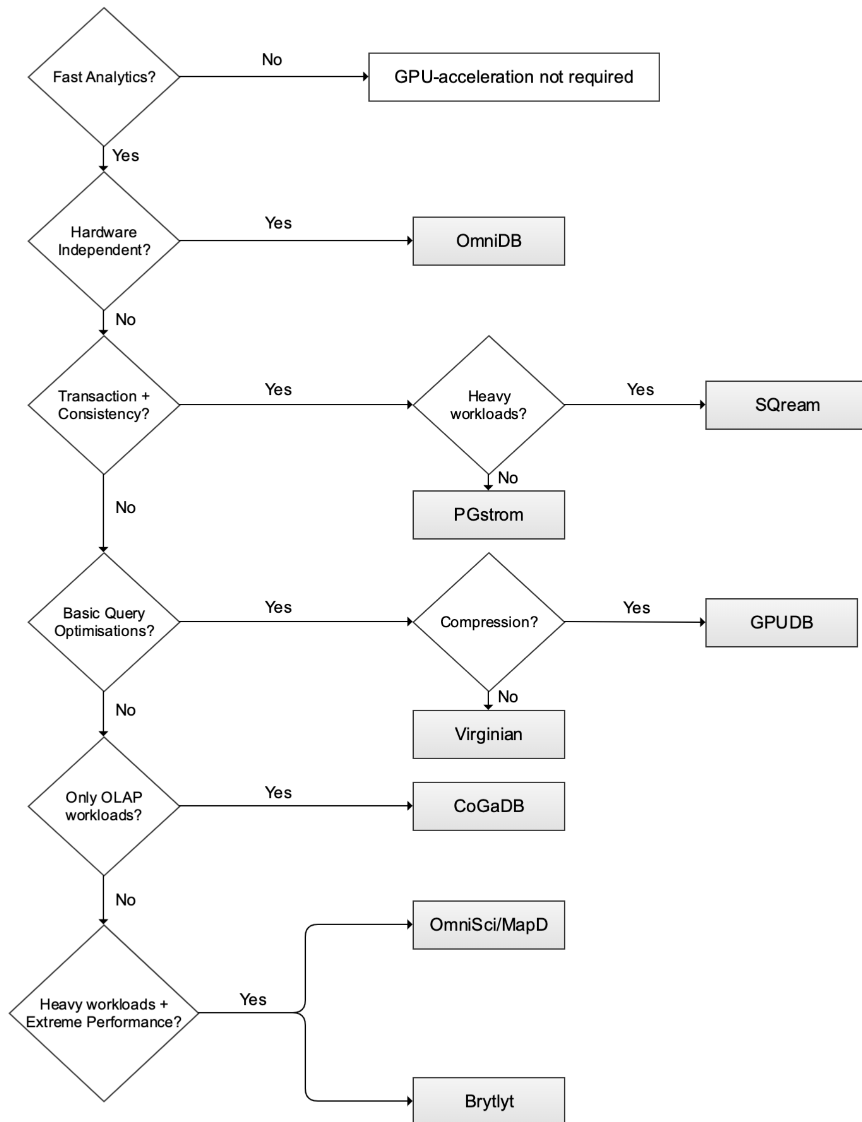

Figure 16: Applicable GPU-accelerated database based on the requirements

References

[1] Shekhar Borkar and Andrew A. Chien. “The Future of Microprocessors”. In: *Commun. ACM* 54.5 (May 2011), pp. 67–77. ISSN: 0001-0782. DOI: 10.1145/1941487.1941507. URL: http://doi.acm.org/10.1145/1941487.1941507.

[2] John D Owens et al. “A survey of general-purpose computation on graphics hardware”. In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[3] Akshay Gautam and Ritesh K Gupta. *GPU enabled database systems*. US Patent 8,392,463. Mar. 2013.

[4] Sebastian Breß et al. “Exploring the Design Space of a GPU-Aware Database Architecture”. In: *New Trends in Databases and Information Systems*. Ed. by Barbara Catania et al. Cham: Springer International Publishing, 2014, pp. 225–234. ISBN: 978-3-319-01863-8.

[5] Sebastian Breß et al. “GPU-Accelerated Database Systems: Survey and Open Challenges”. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events*. Ed. by Abdelkader Hameurlain et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 1–35. ISBN: 978-3-662-45761-0. DOI: 10.1007/978-3-662-45761-0_1. URL: https://doi.org/10.1007/978-3-662-45761-0_1.

[6] André Brodtkorb, Trond Hagen, and Martin Sætra. “Graphics processing unit (GPU) programming strategies and trends in GPU computing”. In: *Journal of Parallel and Distributed Computing* 73 (Jan. 2013), pp. 4–13. DOI: 10.1016/j.jpdc.2012.04.003.

[7] Max Heimel and Volker Markl. “A First Step Towards GPU-assisted Query Optimization”. In: *The Third International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, Istanbul, Turkey* (Jan. 2012).

[8] Sebastian Breß. “The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS”. In: *Datenbank-Spektrum* 14.3 (Nov. 2014), pp. 199–209. ISSN: 1610-1995. DOI: 10.1007/s13222-014-0164-z. URL: https://doi.org/10.1007/s13222-014-0164-z.

[9] Sebastian Breß and Gunter Saake. “Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS”. In: *Proc. VLDB Endow.* 6.12 (Aug. 2013), pp. 1398–1403. ISSN: 2150-8097. DOI: 10.14778/2536274.2536325. URL: http://dx.doi.org/10.14778/2536274.2536325.

[10] Sebastian Breß et al. “Generating Custom Code for Efficient Query Execution on Heterogeneous Processors”. In: *CoRR* abs/1709.00700 (2017). arXiv: 1709.00700. URL: http://arxiv.org/abs/1709.00700.

[11] R. Haberkorn S. Breß and S. Ladewig. *CoGaDB reference manual.* http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/0.3/doc/refman.pdf. 2014.

[12] Bingsheng He et al. “Relational Query Coprocessing on Graphics Processors”. In: *ACM Trans. Database Syst.* 34.4 (Dec. 2009), 21:1–21:39. ISSN: 0362-5915. DOI: 10.1145/1620585.1620588. URL: http://doi.acm.org/10.1145/1620585.1620588.

[13] Pat O’neil, Betty O’neil, and Xuedong Chen. “The Star Schema Benchmark (SSB)”. In: (Jan. 2009).

[14] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. "The Yin and Yang of Processing Data Warehousing Queries on GPU Devices". In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), pp. 817–828. ISSN: 2150-8097. DOI: 10.14778/2536206.2536210. URL: http://dx.doi.org/10.14778/2536206.2536210.

[15] Sriram Padmanabhan et al. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures". In: *Proceedings of the 17th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 567–574. ISBN: 0-7695-1001-9. URL: http://dl.acm.org/citation.cfm?id=645484.656552.

[16] Todd Mostak. "An overview of MapD (massively parallel database)". In: *Massachusetts Institute of Technology* (April 2013). URL: http://www.smallake.kr/wp-content/uploads/2014/09/mapd_overview.pdf.

[17] Todd Mostak. *MapD has rebranded to OmniSci*. Sept. 2018. URL: https://www.omnisci.com/blog/mapd-has-rebranded-to-omnisci/.

[18] Inc. OmniSci. "GPU-accelerated Analytics: Big Data Analytics at Speed and Scale". In: *OmniSci* (Sept 2013). URL: http://www2.omnisci.com/resources/technical-whitepaper/lp.

[19] *Hardware Reference Configuration Guide*. URL: https://www.omnisci.com/docs/v3.6.0/getting-started/hwRefCfgGuide/.

[20] *GPU database acceleration on PowerEdge R940xa – Brytlyt*. Feb. 2019. URL: https://www.brytlyt.com/resources/gpu-database-acceleration-on-poweredge-r940xa/.

[21] *A GPU Database thats just right for you: A Guide*. Feb. 2019. URL: https://www.brytlyt.com/resources/articles/gpu-database-guide/.

[22] *Brytlyt*. URL: https://dbdb.io/db/brytlyt.

[23] CMU Database Group. *Hardware Accelerated Database Lectures #5 - Brytlyt (Richard eyns)*. Nov. 2018. URL: https://www.youtube.com/watch?v=oL0IIMQjFrs&list=PLSE8ODhjZXjbj%20yrcqgE6_lCV6xvzffSN&index=5.

[24] *Brytlyt makes ground-breaking progress on the TPC-H benchmark! – Brytlyt*. Apr. 2019. URL: https://www.brytlyt.com/blog/brytlyt-makes-ground-breaking-progress-on-the-tpc-h-benchmark/.

[25] SQream. *GPU-Accelerated Data Warehouse*. 2018. URL: https://sqream.com/product/architecture/.

[26] Timothy Prickett Morgan. *Telco Calls On GPU-Native SQream SQL Database*. Mar. 2014. URL: http://www.enterpriseai.news/2014/03/28/telco-calls-gpu-native-sqream-sql-database/.

[27] SQream. *SQream DB Technical Whitepaper: A database designed for exponentially growing data*. July 2017. URL: http://temperfield.com/wp-content/uploads/2018/06/SQream-DB-Technical-Whitepaper-Tf.pdf.

[28] PG-Strom. *PG-Strom Overview*. 2019. URL: http://heterodb.com/manual.html#what_is_pgstrom.

[29] Raja K Thaw. *GPU databases;talk of the town*. July 2018. URL: https://www.linkedin.com/pulse/gpu-databasestalk-town-raja-k-thaw/.

[30] Kohei KaiGai. *PG-Strom v2.0 Technical Brief*. Apr. 2018. URL: https://www.slideshare.net/kaigai/pgstrom-v20-technical-brief-17apr2018.

[31] Dimitri Fontaine. *PostgreSQL Concurrency: Isolation and Locking*. July 2018. URL: https://tapoueh.org/blog/2018/07/postgresql-concurrency-isolation-and-locking/.

[32] Shuhao Zhang et al. "OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures". In: *Proc. VLDB Endow.* 6.12 (Aug. 2013), pp. 1374–1377. ISSN: 2150-8097. DOI: 10.14778/2536274.2536319. URL: http://dx.doi.org/10.14778/2536274.2536319.

[33] Peter Bakkum and Srimat Chakradhar. "Efficient Data Management for GPU Databases". In: 2012.

[34] NVIDIA NVLink Fabric. https://www.nvidia.com/en-au/data-center/nvlink/.

[35] Jiang Buxing. *What You Possibly Don't Know About Columnar Storage*. Sept. 2017. URL: https://www.datasciencecentral.com/profiles/blogs/what-you-possibly-don-t-know-about-columnar-storage.

[36] Alastair F. Donaldson et al. "Forward Progress on GPU Concurrency". In: *28th International Conference on Concurrency Theory (CONCUR'17)*. Berlin, Germany, Sept. 2017, 1:1–1:13.