Analysis of GPU-Libraries for Rapid Prototyping Database Operations

A look into library support for database operations

Harish Kumar Harihara Subramanian Bala Gurumurthy Gabriel Campero Durand
David Broneske Gunter Saake
University of Magdeburg
Magdeburg, Germany
firstname.lastname@ovgu.de

***Abstract*—Using GPUs for query processing is still an ongoing research in the database community due to the increasing heterogeneity of GPUs and their capabilities (e.g., their newest selling point: tensor cores). Hence, many researchers develop optimal operator implementations for a specific device generation involving tedious operator tuning by hand. On the other hand, there is a growing availability of GPU libraries that provide optimized operators for manifold applications. However, the question arises how mature these libraries are and whether they are fit to replace handwritten operator implementations not only w.r.t. implementation effort and portability, but also in terms of performance.**

In this paper, we investigate various general-purpose libraries that are both portable and easy to use for arbitrary GPUs in order to test their production readiness on the example of database operations. To this end, we develop a framework to show the support of GPU libraries for database operations that allows a user to plug-in new libraries and custom-written code. Our experiments show that the tested GPU libraries (ArrayFire, Thrust, and Boost.Compute) do support a considerable set of database operations, but there is a significant diversity in terms of performance among libraries. Furthermore, one of the fundamental database primitives – hashing and, thus, hash joins – is currently not supported, leaving important tuning potential unused.

***Index Terms*—GPU-based DBMS, GPU-accelerated DBMS, GPU-libraries, Performance comparison**

I. Introduction

GPUs are common co-processors of a CPU mainly used for offloading graphical computations. Recently, GPUs are also used for offloading general-purpose computations including database operators. In order to get maximum performance, researchers have adapted database operators for GPUs creating a plethora of operator implementations, e.g., group-by [1], [2], selections [3], [4], joins [5], [6], or whole engines [7]–[9].

Developing such tailor-made implementations requires a developer to be an expert on the underlying device [10]. This makes the approach highly time-consuming but leads to the best performance [11]. As an alternative, many expert-written libraries are available that can be included into a system needing only minimal knowledge about the underlying device.

This work was partially funded by the DFG (grant no.: SA 465/51-1 and SA 465/50-1.)

Usually, library operators for GPUs are either written by hardware experts [12] or are available out of the box by device vendors [13]. Overall, we found more than 40 libraries for GPUs each packing a set of operators commonly used in one or more domains. The benefits of those libraries is that they are constantly updated and tested to support newer GPU versions and their predefined interfaces offer high portability as well as faster development time compared to handwritten operators. This makes them a perfect match for many commercial database systems, which can rely on GPU libraries to implement well performing database operators. Some example for such systems are: SQreamDB using Thrust [14], BlazingDB using cuDF [15], Brytlyt using the Torch library [16].

Since these libraries are an integral part of GPU-accelerated query processing, it is imperative to study them in detail. To this end, we investigate existing GPU-based libraries w.r.t. their out-of-the-box support of usual column-oriented database operators and analyze their performance in query execution. Hence, we survey available GPU libraries and focus on the three most commonly used GPU libraries: Thrust, Boost.compute, and ArrayFire to study their support for database operators. Specifically, we explore available operators to determine the library's level of support for database operators and we present which library operators can be used to realize the usual database operators. Using these implementations, we benchmark the libraries based on individual operator performance as well as their execution of a complete query. Overall in this work, we make contributions to the following two directions in order to assess the usefulness of GPU libraries:

- **Usefulness:** We look for libraries with tailor-made implementations for database operators. As a result, we can assess the ad-hoc fit of the libraries for database system implementation (cf. Table II).
- **Usability:** We analyze the performance of the different library-based database operators in isolation as well as for queries from the TPC-H benchmark. This is a key criterion for deciding which library to use for a developer's own database system (cf. Section IV).

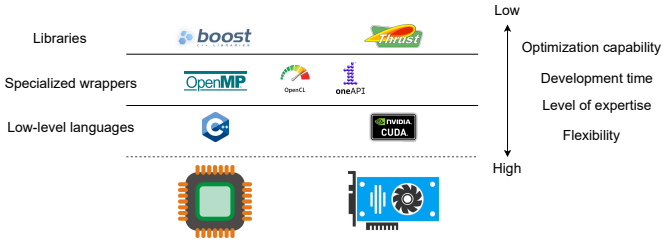

Fig. 1: Hierarchy of abstraction levels characterizing languages, wrappers, and libraries for heterogeneous computing

The paper is structured as follows: In Section II, we classify existing languages and libraries for heterogeneous programming. We review existing GPU libraries and identify how to use them to implement database operators in Section III. In Section IV, we compare the performance of library-based database operators. Finally, we conclude in Section V.

II. Levels of Programming Abstractions

For more than a decade now, the database community has been investigating how to use GPUs for database processing [17]. In fact, the interest for GPU-acceleration is mainly due to the advancements in its processing capabilities as well as the maturity in programming interfaces and libraries. However, for most practitioners it is hard to assess the impact of choosing a specific interface or library. To shed some light on the matter, we compare and review current programming interfaces and libraries. As a result, we broadly categorize them w.r.t. their abstraction level: languages, wrappers and libraries. We place them as a hierarchy since each entity in a level is developed using the lower level constructs. Our Figure 1 shows examples of these identified levels, which we characterize in the following.

Low-Level Languages: At the bottom of the hierarchy, we place device-specific languages. These languages include certain hardware intrinsics, which allows users to access specialized features of the underlying hardware. Such intrinsics are commonly provided by the device vendor and are combined with general purpose programming languages (e.g., C++, ASM). An example of this level are the SSE intrinsics[1] that allow to use SIMD features in modern CPUs [18]. Similar to CPUs, NVIDIA provides its own proprietary API CUDA that provides specialized access to NVIDIA GPUs. For example, CUDA 7.0 and above supports accessing tensor cores. Although these languages grant direct access to the underlying hardware, a developer has to be an expert of the used device architecture to implement a highly optimized operator. Furthermore, changing the device or upgrading to a newer version of the same device might lead to additional rework of the implementations using, for instance, new available intrinsics (e.g., switching from SSE to AVX-512). Therefore, using such low-level languages might improve efficiency but comes with the drawback of high development cost (including a usually large size of program code) and requires expertise on the device features.

Specialized Wrappers: To ease high implementation effort when using low-level languages, wrappers have been developed to hide performance-centric details that a wrapper can handle automatically. To be used, wrapper-specific constructs are added to the source code that will be expanded automatically during execution. One popular example for a wrapper is OpenCL, which offers a set of common constructs for all OpenCL-enabled devices. A program developed using these constructs will be rewritten during compilation based on the target device. Some other examples are: OpenMP [19], Cilk [20] for handling parallelism in CPUs, or oneAPI[2] as Intel's newly pitched wrapper for hardware-oblivious programming on CPUs, GPUs and FPGAs. Although these abstractions significantly reduce the implementation effort compared to low level languages, they are also susceptible to device changes. For example, OpenCL can provide only device portability, but not performance portability [3], [21], [22].

Libraries: At last, there is a plethora of pre-written libraries developed by domain and hardware experts for different devices [23]. Using a library, all internal details of different operator implementations are hidden behind a set of predefined interfaces. Hence, the developer must simply do the right function call based on the underlying scenario. This requires only minimal knowledge on the underlying hardware and implementation. Some examples of libraries include the boost libraries in C++ and the Thrust library for GPUs. Even though these libraries are developed by experts, they are not tailor-made for one underlying use-case. Hence, although a generic implementation of operators suit multiple uses-cases, they can be sub-optimal compared to handwritten use-case-specific implementations. Furthermore, due to the predefined interfaces for operators, one cannot freely combine them for a custom scenario. Instead, we have to chain multiple library calls leading to unwanted intermediate data movements. Thus, libraries provide high productivity in development with only small necessary knowledge about the underlying device (plus, minimal lines of code) but they come with the drawback of potentially sub-optimal performance from the operator implementations.

Used Abstraction Levels in Database Systems: Various GPU-accelerated database systems are developed using the concepts of different levels. Considering low-level languages, GPUQP [17], CoGaDB [7], and the system of Bakkum et al. [4] use CUDA. For wrappers, Ocelot [8], HAWK [24] are implemented in OpenCL. Finally, many commercial database systems use libraries to realize operators, such as SQreamDB [14] or BlazingDB [15], mainly for their robustness and strong vendor support.

Disregarding their low flexibility, libraries give considerable advantages for ad-hoc development of a GPU-accelerated database system reducing its development cost to an acceptable limit. However, with multiple GPU libraries being available,

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[2] https://www.oneapi.com/

the question remains what library has the best support for a rapid prototyping of database operators and which library implementation achieves the best performance.

III. Implementing DBMS Operators With Libraries

In this section, we review different GPU libraries and assess their ad-hoc usability for implementing database operators. To this end, from the selected libraries, we discuss the level of support and the offered functions to realize database operators using these GPU libraries.

A. Review of GPU Libraries

To collect available GPU libraries, we conduct an extensive survey using google, google scholar, and the CUDA website[3]. We look for libraries built on top of the two low-level languages: CUDA (for NVIDIA GPUs) and ROCm (for AMD GPUs), and the two wrappers OpenCL and OneAPI. However, ROCm is itself built over OpenCL, making its performance similar to OpenCL [25]. OneAPI is still in its early stages and not all GPUs are supported at the moment. This shortens our search to libraries over OpenCL and CUDA.

In total, we found 43 libraries that provide GPU-accelerated operators for various domains. The details about the libraries are listed in Table I.

Library	Wrapper/Language	Use case	Reference
AmgX	CUDA	Math	https://developer.nvidia.com/amgx
ArrayFire	CUDA & OpenCL	Database operators	https://developer.nvidia.com/arrayfire
Boost.Compute	OpenCL	Database operators	[26]
CHOLMOD	CUDA	Math	https://developer.nvidia.com/CHOLMOD
cuBLAS	CUDA	Math	https://developer.nvidia.com/cublas
CUDA math lib	CUDA	Math	https://developer.nvidia.com/cuda-math-library
cuDNN	CUDA	Deep learning	https://developer.nvidia.com/cudnn
cuFFT	CUDA	Math	https://developer.nvidia.com/cuFFT
cuRAND	CUDA	Math	https://developer.nvidia.com/cuRAND
cuSOLVER	CUDA	Math	https://developer.nvidia.com/cuSOLVER
cUSPARSE	CUDA	Math	https://developer.nvidia.com/cuSPARSE
cuTENSOR	CUDA	Math	https://developer.nvidia.com/cuTENSOR
DALI	CUDA	Deep learning	https://developer.nvidia.com/DALI
DeepStream SDK	CUDA	Deep learning	https://developer.nvidia.com/deepstream-sdk
EPGPU	OpenCL	Parallel algorithms	[27]
FFmpeg	CUDA	Image and video	https://developer.nvidia.com/ffmpeg
Goopax	OpenCL	Parallel algorithms	https://www.goopax.com/
Gunrock	CUDA	Others - Graph processing	https://github.com/gunrock/gunrock
HPL	OpenCL	Parallel algorithms & Math	https://github.com/fraguela/hpl
IMSL Fortran Numerical Library	CUDA	Math	https://developer.nvidia.com/imsl-fortran-numerical-library
Jarvis	CUDA	Deep learning	https://developer.nvidia.com/nvidia-jarvis
MAGMA	CUDA	Math	https://developer.nvidia.com/MAGMA
NCCL	CUDA	Communication libraries	https://developer.nvidia.com/nccl
nvGRAPH	CUDA	Parallel algorithms	https://developer.nvidia.com/nvgraph
NVIDIA Codec SDK	CUDA	Image and video	https://developer.nvidia.com/nvidia-video-codec-sdk
NVIDIA Optical Flow SDK	CUDA	Image and video	https://developer.nvidia.com/opticalflow-sdk
NVIDIA Performance Primitives	CUDA	Image and video	https://developer.nvidia.com/npp
nvJPEG	CUDA	Image and video	https://developer.nvidia.com/nvjpeg
NVSHMEM	CUDA	Communication libraries	https://developer.nvidia.com/nvshmem
OCL-Library	OpenCL	Database operators	https://github.com/lochotzke/OCL-Library
OpenCLHelper	OpenCL	Others - wrapper	https://github.com/matze/oclkit
OpenCV	CUDA	Image and video	https://developer.nvidia.com/opencv
SkelCL	OpenCL	Database operators & Parallel algorithms	[28]
TensorRT	CUDA	Deep learning	https://developer.nvidia.com/tensorrt
Thrust	CUDA	Database operators	[13]
Triton Ocean SDK	CUDA	Image and video	https://developer.nvidia.com/triton-ocean-sdk
VexCL	OpenCL	Others - vector processing	https://github.com/ddemidov/vexcl
ViennaCL	OpenCL	Math	http://viennacl.sourceforge.net/

TABLE I: Libraries and their properties based on our survey

As GPUs are fundamentally a graphics machine, their parallel processing is perfect for number crunching. Hence, many libraries focus on image processing (7) and math operations (13). Since GPUs were just recently adopted for machine learning workloads[4], only comparatively few libraries are currently available. In case of database operators, libraries that support database operators explicitly are relatively few (only 5) compared to those supporting general vector operations (such as tensor operations offered by VexCL or Eigen tensor).

[3]https://developer.nvidia.com/CUDA-zone

[4]https://developer.nvidia.com/tensor-cores

Database operators	ArrayFire		Boost.Compute		Thrust	
	Support	Function	Support	Function	Support	Function
Selection	+	where(operator())	∼	transform() & exclusive_scan() & gather()	∼	transform() & exclusive_scan() & gather()
Nested-Loops Join	–	–	+	for_each_n()	+	for_each_n()
Merge Join	–	–	–	–	–	–
Hash Join	–	–	–	–	–	–
Grouped Aggregation	+	sumByKey(), countByKey(),	+	reduce_by_key()	+	reduce_by_key()
Conjunction & Disjunction	+	setIntersect(), setUnion()	+	bit_and<T>(), bit_or<T>()	+	bit_and<T>(), bit_or<T>()
Reduction	+	sum<T>()	+	reduce()	+	reduce()
Sort by Key	+	sort()	+	sort_by_key()	+	sort_by_key()
Sort	+	sort()	+	sort()	+	sort()
Prefix Sum	–	–	+	exclusive_scan()	+	exclusive_scan()
Scatter & Gather	–	–	+	scatter(), gather()	+	scatter(), gather()
Product	+	operator*()	+	transform() & multiplies<T>()	+	transform() & multiplies<T>()

+ full support; ∼ partial support; – no support;

TABLE II: Mapping of library functions to database operators

Even from the available libraries, skelCL and OCL-Library are *boilerplates* to OpenCL without any pre-written functions. Therefore, we select Boost.Compute, Thrust and ArrayFire for further analysis built over OpenCL, CUDA and both respectively. Specifically, ArrayFire uses lazy evaluation, Boost.Compute transforms high level functions into OpenCL kernel programs, and Thrust operators are transformed into CUDA C functions.

B. Operator Realization

Since GPUs are predominantly used for column-oriented analytical queries [29]–[31], we consider the operators: projection, (conjunctive) selection, join, aggregation, grouping and sorting (sort-by-key) for our study. Besides these, we also study the parallel primitives: prefix-sum, scatter and gather commonly used for materializing final values. The level of support (i.e., usefulness) and the possible library call for a database operator in the three libraries are listed in Table II. The level of support is determined by the simplicity of the usage of library operators for implementing a database operator. The full support operators have the least interoperability costs and programming effort, because they have a direct functional implementation available in the library. In case of partial support (∼), several function calls are needed to realize an operator. Hence, additional effort is required to pass the intermediate results from one function to another before retrieving the final result. Detailed information on the functional support from these libraries is given in the Function-column of Table II, where we map library functions to the database operators.

C. Summary of Library Usefulness

Overall, compared to ArrayFire, Boost.Compute and Thrust provide an implementation for most of the selected database operators. Specifically ArrayFire does not directly support prefix-sum, nested-loop join, scatter, and gather operations. In terms of functional implementations, it is notable that ArrayFire returns a position list for selections, whereas Thrust and Boost.Compute return bitmaps.

Join implementation: However, all the libraries lack a custom implementation for specialized joins. They lack direct support for hash tables or merge join of sorted results. Hence, these important implementations must be developed from scratch or support needs to be added in the libraries. However, the database community has shown great performance for hash-based joins [32] and, hence, these libraries should be extended by custom operators for hashing in future versions.

IV. Performance Comparison

An important requirement for using library operators for a database system is that they deliver acceptable performance (i.e., usability). Hence, in this section, we study the libraries' performance for different database workloads. We split our evaluation into two main sections: first, we benchmark the performance of individual operators in micro-benchmarks using a synthetic dataset. Afterward, we measure the overall performance of the libraries with complete TPC-H queries.

Experimental setup: All our experiments are conducted on an NVDIA GeForce RTX 2080 Ti with 10GB memory and we use the following library versions: Boost.Compute-v1.71, ArrayFire-v3.7.2, Thrust-v11.0. All these libraries run on top of OpenCL 1.2 and CUDA 10.1 and our evaluation framework is written in C++ and compiled with GCC 9.3.0 running on Ubuntu 18.04[5]. Finally, our synthetic dataset consists of 2^{28} randomly generated integer values unless specified otherwise and the TPC-H dataset is generated with a scale factor 1 (due to size limitations of Boost.Compute).

A. Transfer Time

Since each library has a custom wrapper for accessing the data present in the GPU, they incur different overhead when transferring data to the GPU. Hence, we analyze the data transfer rate of the individual libraries before analyzing the actual operator performance. We test the transfer time with input sizes ranging from 2^{20} integer values (5MB) up to 2^{30} integer values (5GB) and plot it in Figure 2.

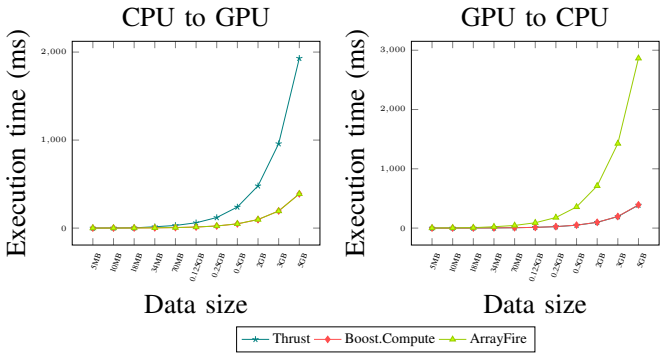

Fig. 2: Transfer times for different libraries

Foremost, the results show a considerable overhead when transferring data to the GPU using Thrust when compared to Boost.Compute or ArrayFire. However while transferring data back, ArrayFire has poor transfer rates. We believe that the additional steps taken by these libraries in allocating / de-allocating the data leads to such poor performance. Even though transfer rates are significant, buffering input columns can easily avoid this overhead. Furthermore, intermediate or final query results that need to be retrieved from the GPU are usually significantly smaller than the input and, hence, this overhead is mostly negligible.

B. Micro-Benchmark: Individual Operators

In this section, we measure the performance impact of operator-specific parameters on the different library implementations. Due to space limitations, we focus on the most common and complex database operators. We exclude sorting, prefix-sum, and map as there are already several papers that analyze the performance of these operators [33], [34].

Selection: As the selection operator is sensitive to the selectivity of the incoming predicate, we evaluate the libraries with varying the selectivity from 1% to 100%. The final result of our selection is the materialized column of matching values. Since Thrust and Boost.Compute create a bitmap and need an additional prefix-sum for materialization, we also show their single performance for creating the bitmap.

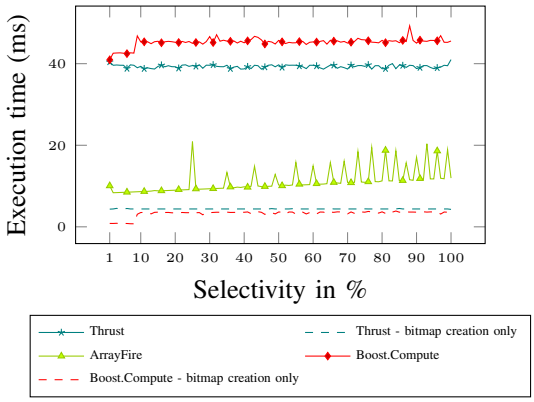

Fig. 3: Performance for selection with varying selectivity

The results in Figure 3 show that the performance of ArrayFire is far better than the performance of Thrust and Boost.Compute for a materialized filtered column. This is because the two libraries need additional steps of a prefix-sum and a gather to arrive at the final result, whereas ArrayFire directly generates position lists from which we can materialize the result. Interestingly, Boost.Compute has the best performance when creating a bitmap, but is the worst when materializing the result. This is due to the bad performing gather, which is consistent to our following results.

As a result, Thrust and Boost.Compute are the best choice for multiple predicates on the same table, because combining bitmaps is faster than intersecting position lists. For single predicates and if subsequent operators work with position lists or materialized columns, ArrayFire should be chosen.

[5]The source code is available here: https://github.com/harish-de/cross_library_execution

Group By: In this experiment, we focus on group-by aggregation where the performance varies according to the spread of groups. We use a uniform distribution of input values and vary the group size from 1% to 100% where 1% has nearly all values belonging to the same group and 100% contains one group per input value.

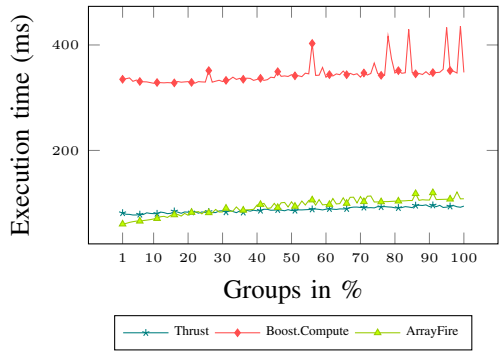

Fig. 4: Performance for Group-by with varying group sizes

The performance in Figure 4 shows that ArrayFire and Thrust have the best performance. Nevertheless, the superior method changes according to the number of groups with ArrayFire performing best for a small amount of groups and Thrust performing best for many groups.

Joins: Joins being a complex operator, requires a considerable time for execution. Our nested loop join uses for_each() - a function to parallelize an operation based on the given input size. Therefore, we measure the performance of the function by varying the cardinality of the left table (|R|) in a range of 2^1 to 2^{19} using a uniform distribution. Varying the input size varies the degree of parallelism, thereby impacting performance. Furthermore, the total size of the right table (|S|) is 2^{28}. In the usual use case, we only join a selection of values with selectivities ranging between 1% and 100%: 1% - 1 out of 100 matches & 100% - all S values match. However, changing selectivity had only minor impact, such that we average our results over all selectivities. In this experiment, we perform our evaluation on Thrust and Boost.Compute only, as ArrayFire does not support joins.

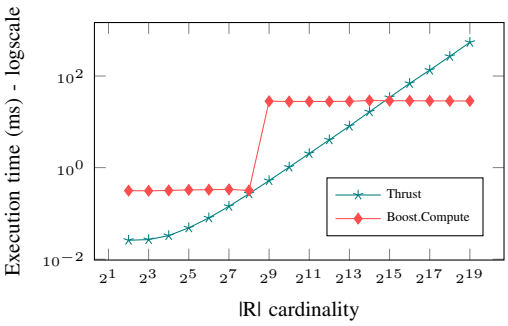

Fig. 5: Performance for join with varying R-table size

We plot the execution time for joins in logscale in Figure 5. From the results, we see that Boost.Compute is comparatively better in terms of parallelization, as its results are linear for a range of inputs. However, even with its linear increasing execution time, Thrust is considerably better for smaller input sizes. In contrast, in case of bigger data sizes, Boost.Compute is superior.

Scatter & Gather: Our final micro-benchmark is to measure the performance of scatter and gather operations, as they are useful in realizing a hashing operation. Hence, we evaluate the performance of scatter and gather giving as positions the results of multiplicative hashing of the input items. We chose multiplicative hashing as it is a function that is commonly used for scattering/gathering keys into hash tables.

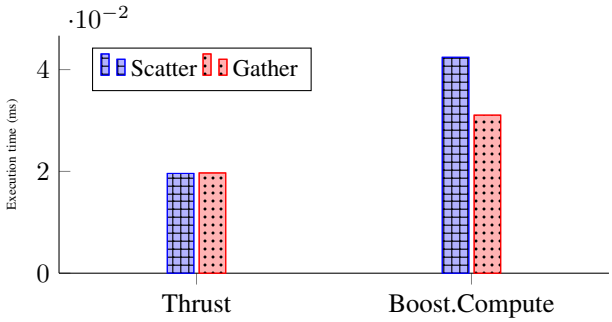

Fig. 6: Performance for scatter & gather

The performance comparison in Figure 6 shows clearly that Thrust has a better scatter and gather time compared to Boost.Compute. Here, the poor performance in Boost.Compute is due to the additional kernel compilation time, whereas Thrust does not have this additional time.

C. TPC-H Performance

In a last comparative experiment, we execute full queries of the TPC-H benchmark (SF=1GB) using our library-based operators. We consider two types of queries based on their most complex operator: group-by (Q1,Q6) and join (Q3,Q4) queries. As ArrayFire does not support a join operation, we have substituted it with that of Thrust in queries Q3 and Q4. Figure 7 depicts only the time taken to execute the operator (as in Table II) and excludes the data transfer time.

For group-by queries, we see that all the libraries have similar performance for grouping. However with conjunctive predicates, Boost.Compute is the fastest followed by Thrust and finally ArrayFire. Since, conjunctive predicates in Boost.Compute and Thrust use bitmaps as intermediate values, conjunction of these predicates are considerably faster. However, Boost.Compute's better selection performance for Q1 is compensated by its bad aggregation performance as we have already seen in Figure 4.

For join-intensive queries, both Boost.Compute and ArrayFire have nearly the same performance. Since, Q3 has two consecutive joins, the runtime of Q3 is nearly twice that of Q4. Overall, the nested-loops join is the main bottleneck compared to all other operations across all the libraries.

In summary, Boost.Compute is better for queries with conjunctive selections, whereas for single predicates Thrust

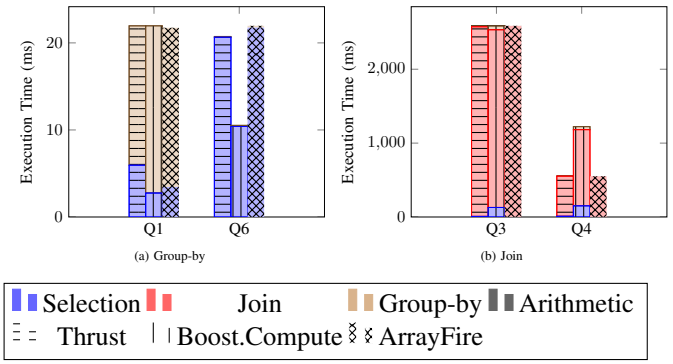

Fig. 7: Performance of TPC-H Queries

or ArrayFire should be used. ArrayFire is better for group-by operations and, finally, the nested-loops join operation is quite expensive on all the libraries.

V. Conclusion

GPUs get more and more often integrated into database processing both academically and commercially. However, building a system from scratch to support database operators is highly time consuming and requires expert knowledge. Therefore, in this work we review different expert-written libraries to be used for faster prototyping a GPU-accelerated database system. Based on our review, we identify 43 GPU libraries out of which 6 support database operators. From these, we study in-depth the support for DBMSs considering the following three libraries built over CUDA and OpenCL: Thrust, Boost.Compute and ArrayFire. Based on our study, we show that not all database operators are supported out-of-the-box by these libraries and one requires additional re-work for operator realization. Our evaluation shows there is no single library that provides the best performance for all supported database operators. Each of the libraries have their own advantages & disadvantages and their functions must be combined in query execution. As a final observation, we see a lack of support for joins from these libraries making the operator the most time-consuming one. For our future work, we would like to extend our work with libraries built on top of other low-level wrappers like OneAPI and do a comprehensive study of all libraries w.r.t. their support for database operators. Furthermore, building an optimizer that chooses the best performing library-based operator during runtime is another important tuning task.

References

[1] T. Karnagel, R. Müller, and G. Lohman, "Optimizing GPU-accelerated group-by and aggregation," in *ADMS*, 2015.

[2] T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl, "SIMD vectorization for hashing in OpenCL," in *EDBT*, 2018, pp. 489–492.

[3] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl, "The operator variant selection problem on heterogeneous hardware," in *ADMS*, 2015, pp. 1–12.

[4] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *GPGPU*, 2010, pp. 94–103.

[5] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious hash-joins on GPUs," in *ICDE*, 2019.

[6] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU join processing revisited," in *DAMON*, 2012, pp. 55–62.

[7] S. Breß, "The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS," *Datenbank-Spektrum*, vol. 14, no. 3, pp. 199–209, 2014.

[8] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *Proc. VLDB Endowment*, vol. 6, no. 9, p. 709–720, Jul. 2013.

[9] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *TODS*, vol. 34, no. 4, pp. 1–39, 2009.

[10] I. Arefyeva, G. Campero Durand, M. Pinnecke, D. Broneske, and G. Saake, "Low-latency transaction execution on graphics processors: Dream or reality?" in *ADMS*, 2018.

[11] D. Broneske, S. Breß, M. Heimel, and G. Saake, "Toward hardware-sensitive database operations," in *EDBT*, 2014, pp. 229–234.

[12] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "CUDPP: CUDA data parallel primitives library," Dec 2016. [Online]. Available: https://github.com/cudpp/cudpp

[13] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*. Elsevier, 2012.

[14] SQream Technologies, "GPU based SQL database," 2010. [Online]. Available: https://sqream.com/product/

[15] BlazingDB, "High Performance GPU Database for Big Data SQL," 2015. [Online]. Available: https://blazingdb.com/

[16] Brytlyt, "World's most advanced GPU accelerated database," 2013. [Online]. Available: https://www.brytlyt.com/

[17] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "GPUQP: Query co-processing using graphics processors," in *SIGMOD*, 2007, pp. 1061–1063.

[18] B. Gurumurthy, D. Broneske, M. Pinnecke, G. C. Durand, and G. Saake, "SIMD vectorized hashing for grouped aggregation," in *ADBIS*, 2018, pp. 113 – 126.

[19] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI*, 1998, pp. 212–223.

[21] M. Moghaddamfar, C. Färber, W. Lehner, and N. May, "Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA," in *DAMON*, 2020.

[22] A. Becher, L. B.G. *et al.*, "Integration of FPGAs in database management systems: Challenges and opportunities," *Datenbank-Spektrum*, 2018.

[23] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM CSUR*, vol. 47, no. 4, pp. 1–35, 2015.

[24] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl, "Generating custom code for efficient query execution on heterogeneous processors," *The VLDB Journal*, vol. 27, no. 6, p. 797–822, Dec. 2018.

[25] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *ISPASS*. IEEE, 2018, pp. 209–218.

[26] J. Szuppe, "Boost.Compute: A parallel computing library for C++ based on OpenCL," *IWOCL*, vol. 15, pp. 1–39, 2016.

[27] O. S. Lawlor, "Embedding OpenCL in C++ for expressive GPU programming," in *WOLFHPC*, 2011.

[28] M. Steuwer, P. Kegel, and S. Gorlatch, "Skelcl-a portable skeleton library for high-level GPU programming," in *IPDPS*, 2011, pp. 1176–1182.

[29] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "GPU-accelerated database systems: Survey and open challenges," *TLDKS*, pp. 1–35, 2014.

[30] I. Arefyeva, D. Broneske, M. Pinnecke, M. Bhatnagar, and G. Saake, "Column vs. row stores for data manipulation in hardware oblivious cpu/gpu database systems," in *GvDB*. CEUR-WS, 2017, pp. 24–29.

[31] M. Pinnecke, D. Broneske, G. C. Durand, and G. Saake, "Are databases fit for hybrid workloads on GPUs? A storage engine's perspective," in *ICDE*, 2017, pp. 1599–1606.

[32] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump up the volume: Processing large data on GPUs with fast interconnects," in *SIGMOD*, 2020, pp. 1633–1649.

[33] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware*, 2007.

[34] D. P. Singh, I. Joshi, and J. Choudhary, "Survey of GPU based sorting algorithms," *Int. J. Parallel Prog.*, vol. 46, no. 6, pp. 1017–1034, 2018.