Distributed and Parallel Databases (2023) 41:489–509
https://doi.org/10.1007/s10619-023-07431-3

Out-of-the-box library support for DBMS operations on GPUs

Harish Kumar Harihara Subramanian[1] · Bala Gurumurthy[1] · Gabriel Campero Durand[1] · David Broneske[2] · Gunter Saake[1]

Accepted: 20 April 2023 / Published online: 10 May 2023

Abstract

GPU accelerated query execution is still ongoing research in the database community, as GPUs continue to be heterogeneous in their architectures varying their capabilities (e.g., their newest selling point: tensor cores). Hence, many researchers come up with optimal operator implementations for a specific device generation involving tedious operator tuning by hand. Alternatively, there is a growing availability of GPU libraries providing optimized operators for various applications. However, the question arises of 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 performance. In this paper, we investigate various general-purpose libraries that are both portable and easy to use for arbitrary GPUs 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 framework allows for easy pluggability of new libraries for query execution using a simple task model. Using this framework, we develop multiple libraries (ArrayFire, Thrust, and boost. compute) supporting many database operations. We use these libraries to experiment with different devices to see the impact of the underlying device. Based on our experiments, we see 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.

Keywords GPU-based DBMS · GPU-accelerated DBMS · GPU-libraries · Performance comparison

✉ Bala Gurumurthy
balasubramanian.gurumurthy@ovgu.de

1 DBSE Group, University of Magdeburg, Universitätsplatz, 39104 Magdeburg, Saxony-Anhalt, Germany

2 German Center for Higher Education Research and Science Studies, 30159 Hannover, Lower Saxony, Germany

1 Introduction

GPUs are common co-processors of a CPU 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 tailor-made implementations requires a developer to be an expert on the underlying device [10]. However, such expert implementations take time to develop but ensures the best performance [11]. As an alternative, many expert-written libraries are available that can be included in an existing system with only minimal knowledge about the underlying device.

These libraries for GPUs are either written by hardware experts [12] or are available out-of-the-box from device vendors [13]. In this work, we survey the existing libraries and identify more than 40 libraries for GPUs each packing a set of operators commonly used in one or more domains. The common benefits of these libraries are that they are constantly being updated to perform the best, repeatedly tested to support newer GPU versions, and their predefined interfaces offer high portability and faster development time compared to handwritten operators. This makes the libraries a suitable match for many commercial database systems to offer GPU support easily. Some examples of systems using libraries for GPU support 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 produce 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 2).
- **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. Sect. 5).
- **Portability** We experiment across two different grades of GPU to see the impact of libraries from the underlying hardware.

The paper is structured as follows: In Sect. 2, 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 Sect. 3. In Sect. 5, we compare the performance of library-based database operators. Finally, we conclude in Sect. 6.

2 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 in 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 Fig. 1 shows examples of these identified levels, which we characterize in the following.

2.1 Low-level languages

At the bottom of the hierarchy, we place device-specific languages. These languages include certain hardware intrinsics, which allow 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.

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

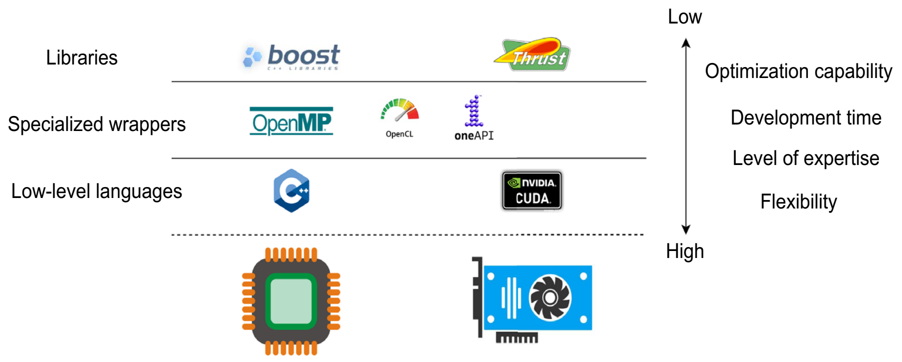

Fig. 1 Hierarchy of abstraction levels characterizing languages, wrappers, and libraries for heterogeneous computing

2.2 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].

2.3 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 suboptimal compared to handwritten use-case-specific implementations. Furthermore, due to the

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

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 suboptimal performance from the operator implementations.

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

3 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 implement database operators using these GPU libraries.

3.1 Review of GPU libraries

To collect available GPU libraries, we conduct an extensive survey using google, google scholar, and the CUDA website.[3] Generally, there are four different frameworks/languages used by libraries over a GPU namely: CUDA, OpenCL, ROCm, and oneAPI. However, ROCm has been not widely adopted and its performance is similar to that of OpenCL [25]. Next, oneAPI is still in its early stages of development and not all GPUs are currently supported [26]. This shortens our search over OpenCL and CUDA. Between these two frameworks, we found 43 libraries that provide GPU-accelerated operators for various domains. The library details are listed in Table 1.

As GPUs are fundamentally graphics machines, their parallel processing is perfect for number crunching. Hence, as shown in Fig. 2 many libraries focus on

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

Table 1 Libraries and their properties based on our survey

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	[27]
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	[28]
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

Table 1 (continued)

Library	Wrapper/Language	Use case	Reference
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	[29]
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/

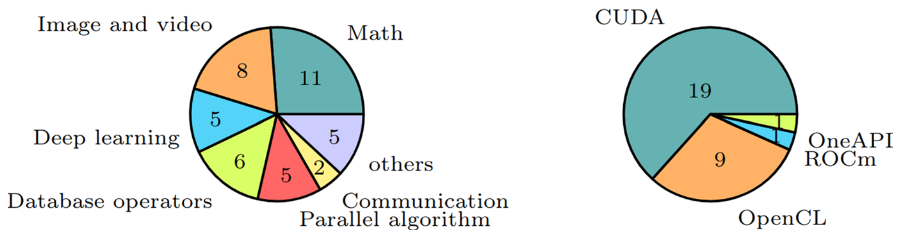

Fig. 2 Proportion of GPU libraries. Left: proportion of libraries across various application domains. Right: Proportion of GPU libraries and their underlying implementation language

image processing (7) and math operations (13). Since GPUs were recently adopted for machine learning workloads,[4] only a few libraries are currently present. With databases, libraries that support database operators explicitly are relatively few (5) compared to those supporting general vector operations (such as tensor operations offered by VexCL or Eigen tensor).

Even from the available libraries, skelCL and OCL-Library are *boilerplates* to OpenCL without any pre-written functions [30]. These have no direct functions available for implementing database operations. Therefore, we select the remaining ones: boost.compute, Thrust, and ArrayFire for further analysis built over OpenCL, CUDA, and both, respectively. Among these, ArrayFire uses lazy evaluation while boost.compute transforms high-level functions into OpenCL kernel programs, and Thrust operators are transformed into CUDA C functions.

3.2 Operator realization

Since GPUs are predominantly used for column-oriented analytical queries [31–33], 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 2. 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 the case of partial support (~), several function calls are needed to implement 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 2, where we map library functions to the database operators.

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

Table 2 Mapping of library functions to database operators

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

3.3 Summary of library usefulness

Overall, when compared to ArrayFire, the other two libraries- boost.compute, and Thrust have multiple alternative implementations for selection. Specifically, ArrayFire does not directly support prefix-sum, nested-loop join, scatter, and gather operations. Regarding functional implementations, it is notable that ArrayFire returns a position list for selections, whereas Thrust and boost.compute return bitmaps.

Join implementation A major limitation is that 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 to the libraries. However, the database community has shown great performance for hash-based joins [34] and, hence, these libraries should be extended by custom operators for hashing in future versions.

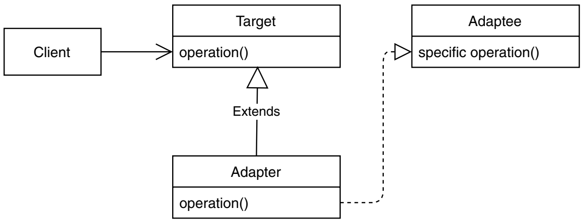

Fig. 3 Adapter design pattern used for adding libraries

4 A connecting framework for library operators

As a next step towards rapid prototyping, it is necessary to execute library operators in a common environment. This is an important step, since we want to assess their runtime without any side effects and also allow for an interoperability between operators of different libraries if the performance difference is significant. In the following, we describe our generalized task model and adapter pattern that we use for interfacing the libraries.

4.1 Task model

Our task model handles the implementation of an operator within a unified interface for all libraries. Currently, our framework supports ArrayFire and Thrust implemented in CUDA, while boost.compute is implemented in C++. Hence, the task model implements cross-platform (CUDA and C++) execution of GPU libraries in a single code base. Furthermore, our framework can include a new library or handwritten code with writing a simple additional wrapper. To support extensibility, we make use of the adapter design pattern. The detail of the programming structure is explained in the next section.

4.2 Adapter pattern

Since the libraries differ in container and operator arguments, our framework needs an easy way to interface with these library operators. A promising feature is that they support the same data type - a vector. In Fig. 3, we depict the adapter design pattern that we use for interfacing the library operators. The idea is that the end-user interacts with the target: an interface without implementation. Each library implementation consists of an adapter and an adaptee. As a result, the adapter bridges the incompatibility between the target and the adaptee. For example, the target of the container is a C++ STL vector, which can be converted into a thrust::device_vector<T>, a boost::compute::vector<T>, and an af::array in corresponding adapters. As a result, we can easily switch between operator implementations of different libraries. To this end, the adapter performs library-specific data conversions and includes library-specific additional arguments.

5 Performance comparison

An essential 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 a commodity - NVIDIA GeForce RTX 2080 Ti with 10 GB memory and server-grade - NVIDIA V100 with 32 GB memory GPU respectively. 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. Our evaluation framework is written in C++ and compiled with GCC 9.3.0 running on Ubuntu 18.04.[5]

Dataset we synthesize datasets for our micro-benchmark. 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 of 10. Note: This is the maximum scale factor up to which the execution across libraries is supported. Any larger scale factors are not executed due to space limitations from boost.compute.

5.1 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 (5 MB) up to 2^{30} integer values (5 GB) and plot it in Fig. 4.

Foremost, the results show a considerable overhead when transferring data to the GPU using Thrust when compared to boost.compute or ArrayFire on both devices. However, while transferring back, ArrayFire shows poor transfer rates. We believe that the additional steps taken by these libraries in allocating / de-allocating the data lead 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. Finally, we see that the transfer rates for V100 are considerably faster than RTX 2080 Ti, even though these two systems use the same PCI-e 3.0 standards for data transfer. However, we see an identical profile for CPU to GPU transfer in both devices - Thrust takes more time to transfer data. We believe this is mainly due to the implementation of the copy operator in this library. Whenever a host-to-device copy is made (which can be achieved using a simple assignment operator, =), it calls CUB's uninitialized_copy() function that allocates data space followed by data transfer, which leads to poor performance.

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

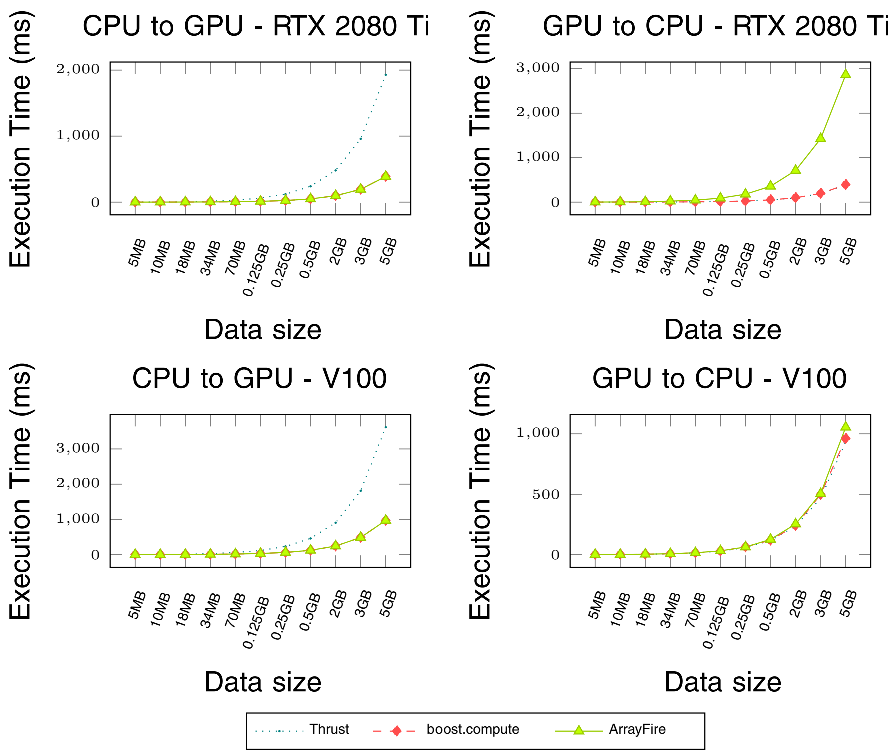

Fig. 4 Transfer times for different libraries

However, when copying results back, it simply does a data copy on the pre-allocated memory in the CPU space.

5.2 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 [35, 36].

5.2.1 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.

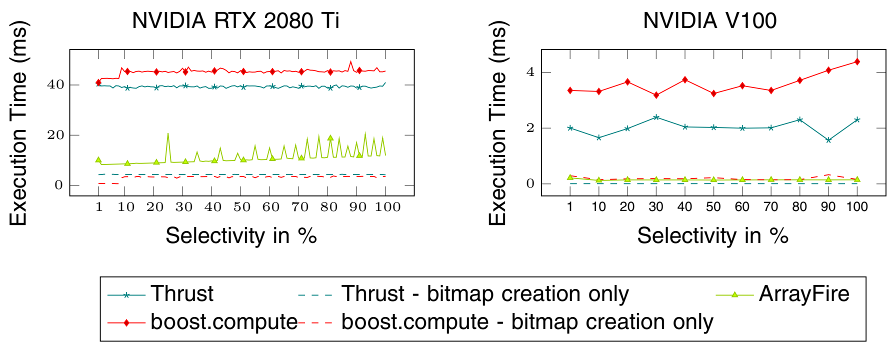

Fig. 5 Performance for selection with varying selectivity

The results in Fig. 5 show that the performance of ArrayFire is far better than the performance of Thrust and boost.compute for a materialized filtered column (solid line) across both devices. The main benefit in ArrayFire is that it can directly generate filtered results without additional prefix-sum and gather steps to arrive at the final results. Instead, ArrayFire generates position lists from which we can directly 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.

5.2.2 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.

The performance in Fig. 6 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. Further, the performance of V100 shows a drop after a group size of 30%. This shows that V100 can manage multiple data writes efficiently when repeatedly accessing a single location.

5.2.3 Joins

Joins being complex operator, generally requires a considerable time for execution. In the case of libraries, we can only support nested loop joins (cf. Table 2). Our

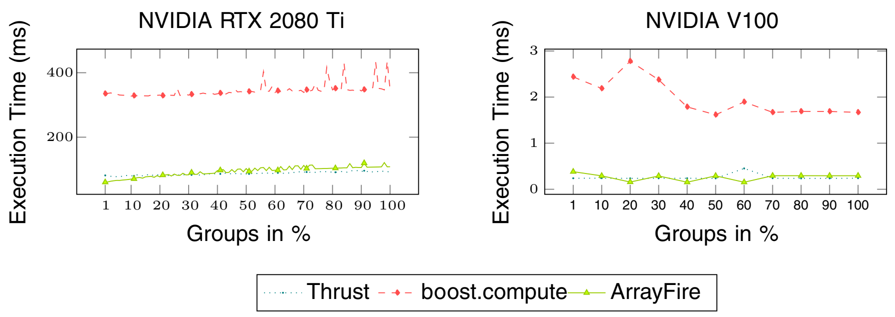

Fig. 6 Performance for Group-by with varying group sizes

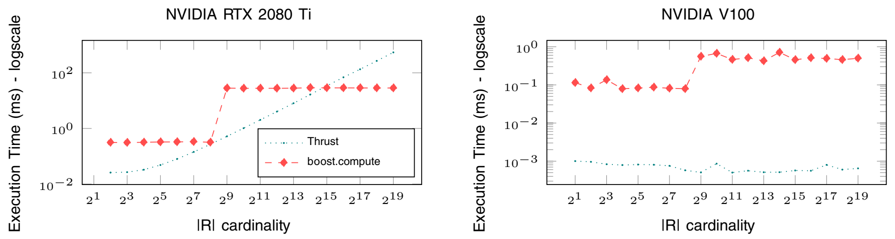

Fig. 7 Performance for join with varying R-table size

nested loop join uses for_each() - a function to parallelize an operation based on the given input size.

We measure the performance of join implementations varying the cardinality of the left table (|R|) in a range of 2^1 to 2^{19} using a uniform distribution.[6] We vary the input size, as it directly impacts the degree of parallelism during the execution of a join. Additionally, the execution also depends on the size of the right-side table. Therefore, we keep the size of the right table (|S|) as 2^{28}. Finally, only Thrust and boost.compute support join operations in the form of a nested loop join. Since ArrayFire does not offer a custom for_each() function, it is not part of the evaluation.

We plot the execution time for joins over the two devices in logscale in Fig. 7. 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 for RTX 2080 Ti. 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. Considering the results on the V100, Thrust is clearly the winner as we can see a huge difference in the runtime of the two libraries in Fig. 7.

Furthermore, results pertaining to boost.compute show a near-constant growth in performance. This is mainly due to the way the execution spawns threads for executing the custom function. Until a data size of 2^9, the framework uses a different

[6] Note that 2^{19} is the maximum data size until which we get a reasonable execution time.

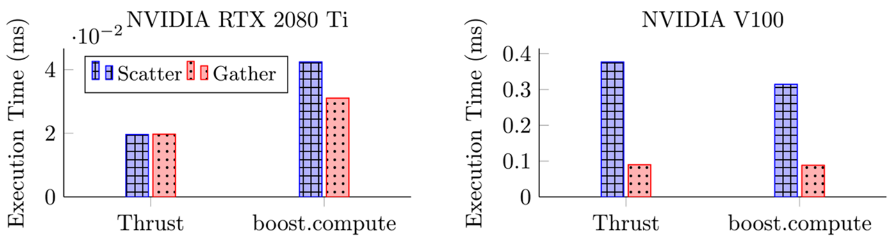

Fig. 8 Performance for scatter and gather

number of threads to spawn, while for any data size greater they process the data using multiple iterations.

5.2.4 Scatter and 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.

The performance comparison in Fig. 8 shows clearly that Thrust has a better scatter and gather time compared to boost.compute for RTX 2080 Ti. Here, the poor performance in boost.compute is due to the additional kernel compilation time, whereas Thrust does not have this additional time. Considering the V100 results, we see that scatter operations take longer than gather operations for both libraries. Such poor behavior for scatter shows the overhead of executing random memory accesses on global memory. This behavior indirectly represents the bottleneck in the memory controller of V100 in resolving memory accesses to the global memory.

5.2.5 Summary

Overall, we see that there is no one good library that gives consistent performance benefits for all database operations. For selection, ArrayFire is the clear winner being nearly 2x faster than the other libraries. In the case of group-by, both Thrust and ArrayFire perform similarly. In case of joins, boost.compute gives constant performance across various table sizes; however Thrust gives a better performance compared to boost.compute in commodity GPUs. Finally, with scatter and gather operations, Thrust's performance is better in commodity GPU and both Thrust and boost.compute behave the same in server-grade GPUs. Based on these results, we can now devise the execution of TPCH queries with a single library as well as cross-library calls.

5.3 TPC-H performance

Extending the previous experiments with individual operators, in this section we use the operator implementations to execute complete queries. We use the TPCH dataset with scale factor 10 (SF 10) for executing two query types: group-by (Q1, Q6) and join (Q3, Q4) queries. In the case of the join queries, we substitute ArrayFire with Thrust implementation as the former does not support joins. Finally, we experiment with two different scenarios: single-library and cross-library executions. The results from the execution are explained in the sections below.

5.3.1 Single library performance

In this experiment, we execute TPCH queries using single homogeneous library calls. The resultant execution time across the considered devices is depicted in Fig. 9. The results depict only the time taken to execute the operator (as in Table 2) and exclude the data transfer time into the device memory.

Group-By Depending on the cardinality and complexity of the operator clauses (like multiple group-by or multiple conjunctive predicates), the execution characteristics vary. This is evident from the results of Q1 with more time invested in computing group-by aggregates whereas Q6 has most of its time spent in the selections. Though the rank of the fast-performing libraries remains the same across RTX 2080 Ti and V100, there is a significant difference in their performance profile. Specifically, we see that ArrayFire performs significantly better in V100. This is in accordance with its performance difference from group-by experiments (cf. Fig. 6). Since the V100 is equipped with much more cuda cores, a larger data size fits the execution in the device and more aggregates are resolved at a given timespan. Overall, we see that boost.compute performs well with selection operations whereas ArrayFire works well with group-by aggregation.

We see that with Q1 in Fig. 9c, ArrayFire shows an improved performance for group-by aggregation compared to other approaches. However, the throughput profile for Q6 remains the same for the libraries. Still, we see an increase in execution time for group-by aggregation on Thrust and boost.compute for the device. We believe such behavior is due to hardware sensitivity during execution. Specifically, the longer group-by duration is again from the random access to the global memory while grouping the input. As we have seen in Fig. 8 for V100, random access to global memory is a bottleneck, which is also the case with group-by queries here.

Join As we have seen earlier, joins are considerably more expensive than other database operations when executed using libraries. This is also reflected in the query results. Almost 90% of the overall time is invested in executing join operations. Unlike with group-by queries, changing devices reflects in the performance profile across the libraries. This is again mainly due to the increase in CUDA cores in V100. Overall, we see that V100 increases the performance by about 10x compared to RTX 2080 Ti. Clearly, we see that libraries are not a suitable solution for executing join operations. The current solutions in research [37, 38] are faster than the

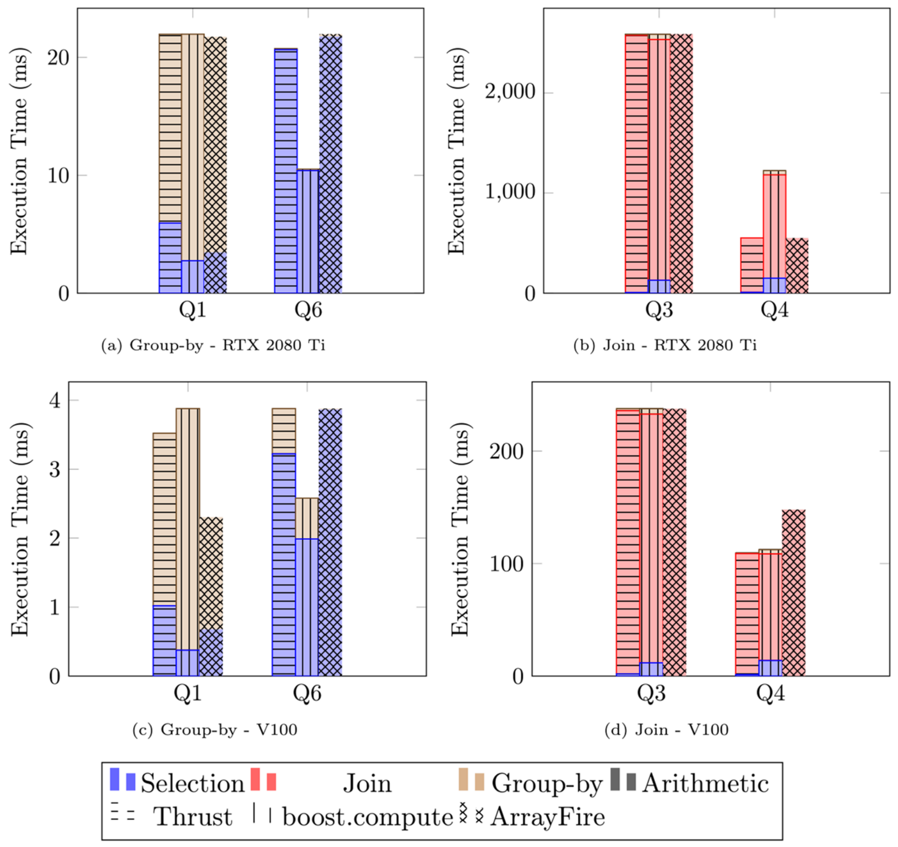

Fig. 9 Performance of TPC-H queries

naive library counterparts. However, we also believe this is mainly due to the lack of support for more sophisticated join algorithms like hash joins and sort-merge joins.

The execution over V100 also has the same performance characteristics as above, except, Q4 shows differences in execution time compared to RTX 2080 Ti. Here, ArrayFire performs poorly compared to other libraries. Again, we believe the difference in performance arises due to the penalty of random access in V100.

Summary For group-by queries over RTX 2080 Ti (cf. Fig. 9a, we see a similar performance across libraries. 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, the conjunction of these predicates is considerably faster. However, boost.compute's better selection performance for Q1 is compensated by its bad aggregation performance, as we have already seen in Fig. 6. However, this is not the case with the V100 device. Overall, boost.compute is better for queries with conjunctive selections, whereas for single predicates Thrust 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. Finally, the libraries are also sensitive to the underlying device (even to the generations).

5.3.2 Cross library performance

As our final experiment, we evaluate the combined performance of executing TPCH queries across various libraries. To this end, we evaluate the two TPCH queries Q1 and Q3 as samples for group-by and join queries respectively. The execution plan considers the best-performing library for the different operators in the query.

From our previous experiments, we identify that selection is best executed with boost.compute and join & group-by with Thrust. Hence, a reasonable goal is to enable cross-library execution by using our adapter pattern, which we described in Sect. 4.2. Additionally, using mixed libraries for execution also introduces the overhead of translating data from one library format to another. However, this overhead is negligible, as we switch the logical data format instead of physically moving the data. The result of TPCH execution with this mixed library execution is given in Fig. 10. The overall performance, when compared with the ones in Fig. 9, shows a considerable decrease in execution time. A decrease of around 25% for Q1 and Q6 can be observed while for join queries, the improvement is not that significant due to the overhead of executing joins.

6 Conclusion

GPUs are 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 of 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 DBMS 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 has its own advantages & disadvantages and their functions must be combined in query execution. we see a lack of support for joins from these libraries making the operator the most time-consuming one. As a final observation, we see a change in the performance of the libraries across different GPU generations. Based on our observations, we conclude the following:

- **Usefulness** The usefulness of libraries for DBMS is fairly restrictive. Not all database operations are supported out-of-the-box through these libraries. Dur-

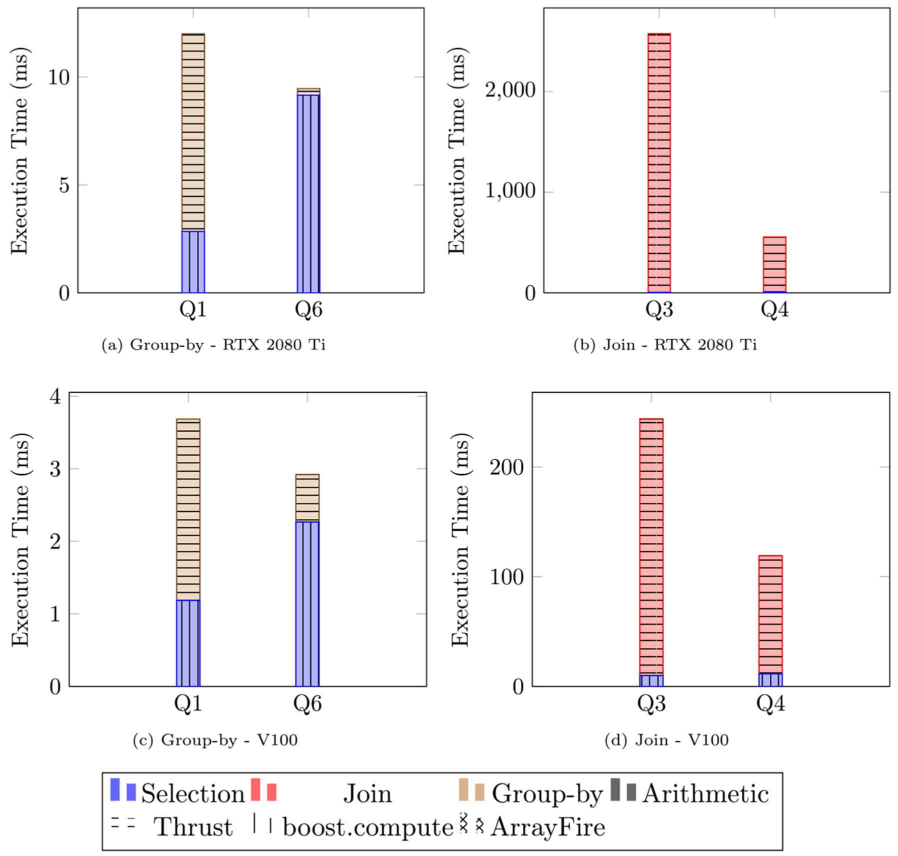

Fig. 10 Performance of TPC-H queries using inter-library execution

ing our study, we especially identified hash-based or sort-based joins to be a pain point, which calls for future work in library implementations.

- **Usability** Based on our evaluations, not all library functions are performance efficient. Apart from the obvious deficiency of joins, also the performance of other operators across libraries varies heavily. Hence, to reach the best performance, users would need to test all libraries and combine their operators based on the query. Since interfacing between libraries is still manual work, future work needs to create a solution for inter-library execution and automatic library/operator selection.
- **Portability** Libraries can be executed across various devices out-of-the-box with fewer rework. However, our evaluation shows that new devices have a different performance profile for the same operator. Hence, this poses another challenge to the library/operator selection problem.

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.

Acknowledgements This work was partially funded by the DFG (grant no.: SA 465/51-2 and PI 447/9-2).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Karnagel, T., Müller, R., Lohman, G.: Optimizing GPU-accelerated group-by and aggregation. In: ADMS (2015)
2. Behrens, T., Rosenfeld, V., Traub, J., Breß, S., Markl, V.: SIMD vectorization for hashing in OpenCL. In: EDBT, pp. 489–492 (2018)
3. Rosenfeld, V., Heimel, M., Viebig, C., Markl, V.: The operator variant selection problem on heterogeneous hardware. In: ADMS, pp. 1–12 (2015)
4. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. In: GPGPU, pp. 94–103 (2010)
5. Sioulas, P., Chrysogelos, P., Karpathiotakis, M., Appuswamy, R., Ailamaki, A.: Hardware-conscious hash-joins on GPUs. In: ICDE (2019)
6. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.B.: GPU join processing revisited. In: DAMON, pp. 55–62 (2012)
7. Breß, S.: The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. Datenbank-Spektrum **14**(3), 199–209 (2014)
8. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. Proc. VLDB Endow. **6**(9), 709–720 (2013)
9. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. TODS **34**(4), 1–39 (2009)
10. Arefyeva, I., Campero Durand, G., Pinnecke, M., Broneske, D., Saake, G.: Low-latency transaction execution on graphics processors: dream or reality? In: ADMS (2018)
11. Broneske, D., Breß, S., Heimel, M., Saake, G.: Toward hardware-sensitive database operations. In: EDBT, pp. 229–234 (2014)
12. Harris, M., Owens, J., Sengupta, S., Zhang, Y., Davidson, A.: CUDPP: CUDA Data Parallel Primitives Library. https://github.com/cudpp/cudpp. Accessed 25 Jan 2021
13. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for CUDA (2012)
14. SQream Technologies: GPU based SQL database (2010)
15. BlazingDB: High Performance GPU Database for Big Data SQL (2015)
16. Brytlyt: World's most advanced GPU accelerated database (2013)
17. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N.K., Luo, Q., Sander, P.V.: GPUQP: query co-processing using graphics processors. In: SIGMOD, pp. 1061–1063 (2007)
18. Gurumurthy, B., Broneske, D., Pinnecke, M., Durand, G.C., Saake, G.: SIMD vectorized hashing for grouped aggregation. In: ADBIS, pp. 113–126 (2018)

19. Chandra, R., Dagum, L., Kohr, D., Mayden, D.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, New York (2008)
20. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI, pp. 212–223 (1998)
21. Moghaddamfar, M., Färber, C., Lehner, W., May, N.: Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA. In: DAMON (2020)
22. Becher, A., B.G., L., et al.: Integration of FPGAs in database management systems: challenges and opportunities. Datenbank-Spektrum (2018)
23. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM CSUR **47**(4), 1–35 (2015)
24. Breß, S., Köcher, B., Funke, H., Zeuch, S., Rabl, T., Markl, V.: Generating custom code for efficient query execution on heterogeneous processors. VLDB J. **27**(6), 797–822 (2018)
25. Sun, Y., Mukherjee, S., Baruah, T., Dong, S., Gutierrez, J., Mohan, P., Kaeli, D.: Evaluating performance tradeoffs on the Radeon open compute platform. In: ISPASS, pp. 209–218 (2018). IEEE
26. Ashbaugh, B., Bader, A., Brodman, J., Hammond, J., Kinsner, M., Pennycook, J., Schulz, R., Sewall, J.: Data parallel c++ enhancing sycl through extensions for productivity and performance. In: Proceedings of the International Workshop on OpenCL, pp. 1–2 (2020)
27. Szuppe, J.: Boost.Compute: a parallel computing library for C++ based on OpenCL. In: IWOCL 15, pp. 1–39 (2016)
28. Lawlor, O.S.: Embedding OpenCL in C++ for expressive GPU programming. In: WOLFHPC (2011)
29. Steuwer, M., Kegel, P., Gorlatch, S.: Skelcl-a portable skeleton library for high-level GPU programming. In: IPDPS, pp. 1176–1182 (2011)
30. Steuwer, M., Kegel, P., Gorlatch, S.: Skelcl-a portable skeleton library for high-level GPU programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 1176–1182 (2011). IEEE
31. Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: GPU-accelerated database systems: survey and open challenges. In: TLDKS, pp. 1–35 (2014)
32. Arefyeva, I., Broneske, D., Pinnecke, M., Bhatnagar, M., Saake, G.: Column vs. row stores for data manipulation in hardware oblivious CPU/GPU database systems. 24–29 (2017)
33. Pinnecke, M., Broneske, D., Durand, G.C., Saake, G.: Are databases fit for hybrid workloads on GPUs? A storage engine's perspective. In: ICDE, pp. 1599–1606 (2017)
34. Lutz, C., Breß, S., Zeuch, S., Rabl, T., Markl, V.: Pump up the volume: processing large data on GPUs with fast interconnects. In: SIGMOD, pp. 1633–1649 (2020)
35. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Graphics Hardware (2007)
36. Singh, D.P., Joshi, I., Choudhary, J.: Survey of GPU based sorting algorithms. Int. J. Parallel Prog. **46**(6), 1017–1034 (2018)
37. Rui, R., Li, H., Tu, Y.-C.: Join algorithms on GPUs: A revisit after seven years. In: 2015 IEEE International Conference on Big Data (Big Data), pp. 2541–2550 (2015). IEEE
38. Paul, J., He, B., Lu, S., Lau, C.T.: Revisiting hash join on graphics processors: a decade later. Distrib. Parallel Databases **38**, 771–793 (2020)