Hardware-conscious Query Processing in GPU-accelerated Analytical Engines

Periklis Chrysogelos⋆ Panagiotis Sioulas⋆ Anastasia Ailamaki⋆ ‡

⋆École Polytechnique Fédérale de Lausanne ‡RAW Labs SA

firstname.lastname@epfl.ch

ABSTRACT

In order to improve their power efficiency and computational capacity, modern servers are adopting hardware accelerators, especially GPUs. Modern analytical DMBS engines have been highly optimized for multi-core multi-CPU query execution, but lack the necessary abstractions to support concurrent hardware-conscious query execution over multiple heterogeneous devices and take full advantage of the available accelerators.

In this work, we present a Heterogeneity-conscious Analytical query Processing Engine (HAPE), a hardware-conscious analytical engines that targets efficient concurrent multi-CPU multi-GPU query execution. HAPE decomposes heterogeneous query execution into: 1) efficient single-device and 2) concurrent multi-device query execution. It uses hardware-conscious algorithms designed for single-device execution and combines them into efficient intra-device hardware-conscious execution modules, via code generation. HAPE combines these modules to achieve concurrent multi-device execution by handling data and control transfers.

We validate our design by building a prototype and evaluate its performance using radix-join co-processing and the TPC-H benchmark [11]. We show that it achieves up to 10x and 3.5x speed-up on the join against CPU and GPU alternatives and 1.6x-8x against state-of-the-art CPU- and GPU-based commercial DBMS on the queries.

1. INTRODUCTION

Traditionally, analytical DBMS engines have relied on the exponential increase of CPU performance in order to keep up with the also exponential data growth. Initially, CPUs relied on Dennard scaling, improving their performance by increasing their clock frequency. However, after 2005, but this was no longer feasible due to the power wall. As a response, CPU vendors started increasing the core count, thus signaling the beginning of the multi-core era. Now, due to the power wall, the power inefficiency of general-purpose hardware is causing modern servers to change. The increased performance per watt of specialized hardware, such as GPUs, has resulted in their adoption in emerging servers, as can be seen by the almost linear increase over the past decade of accelerator-enabled servers in the TOP500 list. In addition, architects explore designs that go beyond the classical system-wide cache-coherence in favor of increased core scalability.

In order for analytical DBMS engines to scale over time by hardware improvements, they have to efficiently use the heterogeneous hardware of emerging servers. On the CPU front, state-of-the-art engines are using algorithms [29, 30, 6, 27] that match the CPU micro-architecture. Techniques like *vector-at-a-time* execution [7] and *just-in-time code generation* [21, 20] are used to reduce the query execution overheads, while the Exchange [13] operator and HyPer's Morsels [22] are used to parallelize query execution in multi-core and multi-CPU configurations. On the GPU front, recent work has both optimized algorithms for GPU execution [18, 28, 15, 19, 31] as well as investigate the GPU query execution models [33, 14, 24, 8]. The majority of these works does not consider query execution over heterogeneous devices, for example multiple GPUs, and many of them ignore the processing power available in the server's CPUs. The ones that support both, use a high-level framework and/or hardware-oblivious algorithms and thus achieve sub-optimal per-device execution. Lastly, works that support heterogeneous devices, consider only a single type of device per query [25] due to the lack of cross-heterogeneous-device abstractions and algorithms or rely on full wasteful materialization [33, 16, 8].

In this work, we describe a new DBMS engine design for efficient analytical query execution on a heterogeneous multi-CPU multi-GPU server node that combines hardware-conscious algorithms with efficient intra- and inter-device execution models.

Contributions. The contributions of this work are the following:

- We make the case for heterogeneity- and hardware-conscious analytical engines and present HAPE, a design for concurrent execution on heterogeneous hardware that achieves performance of hardware-conscious algorithms on all of them.
- We show that decoupling inter- from intra-device operator design can decrease the design space and at the same time achieves state-of-the-art performance in each device and allows scaling existing algorithms to heterogeneous hardware.
- We evaluate our design by extending Proteus [20, 10] with a GPU-optimized join [31] and its co-processing scheduling to show the effect of hardware-conscious algorithms during hybrid query execution. We show that execution of a GPU radix-join can achieve up to 10x and 3.5x speed up relative to hardware-conscious CPU and GPU joins, respectively. In addition, we show that hybrid query execution achieves 1.6x-8x speed-up on execution time of TPC-H queries against CPU- and GPU-based state-of-the-art DBMS system.

9th Biennial Conference on Innovative Data Systems Research (CIDR '19)
January 13-16, 2019 , Asilomar, California, USA.

With our design, we allow hardware-conscious algorithms for different devices to be combined to achieve efficient concurrent execution over all the compute units available on a multi-CPU, multi-GPU server. HAPE achieves near optimal co-processing performance by combining algorithms optimized for homogeneous systems, effectively avoiding the need to develop algorithms specialized for heterogeneous systems.

2. BACKGROUND

In this section, we discuss hardware-conscious operator algorithms and parallel execution of query plans. In the rest of the paper we will use these components as our building blocks for the heterogeneous hardware-conscious analytical engines.

2.1 Hardware-conscious Operators

While hardware-oblivious algorithms simplify the optimization process and the execution over heterogeneous hardware, tuning algorithms for the underlying hardware can produce significant performance benefits. For modern CPUs, most previous studies take three architectural characteristics into account: cache hierarchy, TLB and SIMD instructions. These dimensions are analyzed in conjunction with the available memory bandwidth and latency.

Prior work has introduced hardware-conscious variants of several operators. including scan-like operators, sort-based operations and index scans [34, 26, 17]. As a heavyweight operator, the join has been studied and tuned extensively for modern CPUs, resulting in multiple variants of the radix hash-join [30, 6, 3, 2, 29]. Specifically, Shatdal et al. [30] proposed a cache conscious variant that introduces a partitioning step. The two input tables are co-partitioned such that for each partition pair the hash table fits in cache. Then, all hash-table accesses during the probing phase are in cache and cache misses are averted. Boncz et al. [6] extend the work to consider TLB misses due to a high number of output partitions and advocate for the use of multiple partitioning passes, each producing a smaller amount of partitions, to avoid this performance pitfall. Schuh et al. [29] note that the common denominator of these studies is that they try to minimize the effects of random memory accesses by minimizing cache and TLB misses. Still, Blanas et al. [5] argue in favor of a hardware-oblivious implementation of hash-joins as they require less parameter tuning and can outperform hardware-conscious implementations in some scenarios.

Compared to CPUs, modern GPUs have a significantly different micro-architecture, including for all three of the aforementioned characteristics. First of all, GPUs depart from the linear hierarchy of CPUs and adopt a fatter cache hierarchy, with a hardware-managed L1 cache, the shared memory, which is a software managed scratchpad, and special caches. Karnagel et al. [19] experimentally showed that GPU TLBs opt for 2MB pages to support the high number of threads and pack more addressable space per TLB entry. Finally, in the GPU SIMT model, each GPU thread has an independent register file but, in contrast with the SIMD model, thread divergence is handled in hardware. Similarly to the CPU case, considering the hardware while designing the algorithms can provide significant improvements. [19] achieve up to 13x speedup for hash-based group by operations. [28] and [18] present partitioned hash-join implementations that use the shared memory for storing histograms and hash tables of partitions.

A limiting factor for GPU algorithms is GPU memory size. Prior works make simplifying assumptions about the types of workloads handled; [28] only addresses the case that at least one of the tables fits in GPU memory. Kaldewey et al. [18] use Unified Virtual Addressing (UVA), to join arbitrarily large data by accessing data over the interconnect. Still, interconnect's bandwidth is an order of magnitude smaller than that of the GPU memory and performance deteriorates for multi-pass algorithms such as radix joins.

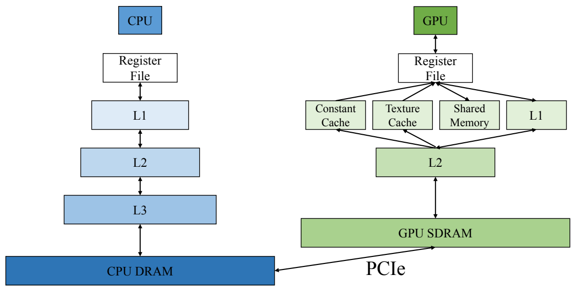

Figure 1: CPU and GPU hierarchy of data caches.

Inter-device co-processing can reduce unnecessary interconnect traffic. Stehle and Jacobsen [32] present an efficient sorting algorithm that consists of two steps: generating sorted runs in GPU and merging them in CPU. Merging in the CPU side allows for a single pass, per direction, over the scarcest resource, the interconnect. Sioulas et al. [31] exploited the high CPU memory-bandwidth to pre-partition the inputs of a partition-based hash-join before sending them to the GPU. The pre-partitioning breaks down big relations into partitions that fit in the GPU memory, while its small fan-out allows for a high throughput in the CPU side. In the GPU side they further partition the inputs to fit the final partitions in the scratchpad and minimize the effect of random accesses.

In Section 4.1, we use this join as a representative example to discuss a hash-join optimized for GPU hardware with respect to the memory hierarchy and compare it with a hardware-oblivious GPU implementation as well as CPU algorithms. In addition, in Section 5 we show how their out-of-GPU execution strategy can be generalized in order to mix different intra-device algorithms to attain efficient multi-device execution.

2.2 Query execution models

In-memory analytical query execution engines traditionally used either a tuple-at-a-time or an operator-at-a-time execution model and suffered from high interpretation overheads and materialization costs, respectively. To amortize these costs, vector-at-a-time [7] execution and just-in-time code generation [21] engines emerged. The former communicates blocks of data at-a-time between operators and is based on the trade-off between interpretation and materialization costs. Usual optimizations involve the use of SIMD instructions and tuning for cache locality. The latter generates specialized code for each query. Intermediate results are passed across operators in registers within tight loops until an operator forces a materialization point. Unlike previous models, overheads are independent of the size of intermediate results.

GPU analytical query execution has similar challenges and techniques. Several GPU systems have used operator-at-time as the execution model [33, 16, 8]. The model is restricted by the GPU memory size and thus is often combined with transferring intermediate result to CPU memory [16, 8]. However, this design causes excessive interconnect traffic, as all the results have to pass over it. In order to reduce the materialization overhead, Paul et al. [24] pipeline data between operators running as separate kernels through OpenCL's communication channels. HorseQC [12] used a block-at-a-time approach and materialized intermediate results in GPU memory to significantly reduce the execution time. In addition, HorseQC and MapD [23] used just-in-time code generation to fuse multiple operators in a single kernel to reduce result materialization and the number of required passes.

The emergence of systems with multiple processors has motivated parallel query execution. On the one hand, the exchange operator [13] has been used to encapsulate parallelism and allow parallel execution without modifying the exiting, single-threaded, operators. On the other hand, Hyper [22] exposes the operators to parallelism, propagating the responsibility of correctly maintaining the data structures to the operators, as for its hash-join which has to guarantee that the hash-table is correctly built using multiple threads. In the heterogeneous context, Voodoo [25] allowed query execution on CPUs and GPUs in MonetDB, but without support for concurrent CPU–GPU execution, load balancing or data structures. Similarly, TVM [9] focused on deep learning workloads and targeted multiple types of devices but considered execution on single-device at a time.

The architecture of modern servers introduces new challenges for targeting multiple types of devices at the same time. Both the exchange and Hyper's approach rely on low-latency system-wide cache coherent memory for synchronization and atomic primitives as well as shared data structures, which is generally lacking in heterogeneous servers. In addition, different devices may have different access rights for different regions of the aggregate memory of the system, based on the system topology as well as the type of devices. To avoid complicating the relational operators and to increase the applicability of our design to future architectures, the HAPE needs to decouple the development of relational operators from the complexities of heterogeneous servers. Our parallelization strategy builds upon the ideas of HetExchange [10], a framework that allows multi-CPU, multi-GPU query execution by encapsulating the heterogeneous parallelism of the server. While HetExchange provides a framework for hardware-oblivious operators, HAPE provides server-wide hardware-conscious execution by composing per-device hardware-conscious algorithms.

TVM automated the optimization of low-level programs to different hardware via an iterative process: a scheduler proposes optimized versions of the input program and the measured performance is used to refine a machine learning model that predicts the performance of the device. TVM can be incorporated in our system to tune the query optimizer as well as the compiler optimizations used by the different device back-ends.

In addition, different devices are fit for different workloads and can be leveraged synergistically. Appuswamy et al. [1] propose the archipelago abstraction which encapsulates a set of devices and a target workload as a means to partition resources per functionality. Our work focuses on the design of a hardware-conscious analytical multi-CPU, multi-GPU archipelago.

3. THE CASE FOR HAPE

Heterogeneity-conscious Analytical query Processing Engines (HAPE) allow DBMS systems to take advantage of heterogeneous hardware present in modern servers by 1) encapsulating heterogeneity and multi-device parallelism, 2) providing a unified execution model and 3) embracing single-device hardware-conscious operators. These operators are then composed together to provide server-wide hardware-conscious execution, while the encapsulation handles their communication and synchronization.

Decoupling heterogeneity from execution. HAPE exploits the observation that by encapsulating the inter-device functionality, the remaining system is composed by single (homogeneous) device subsystems. HAPE minimizes the effect of heterogeneity and allows the rest of the system to be build by combining existing work on homogeneous systems. Operators specialized to each micro-architecture may be used for each type of device. Just-in-time code

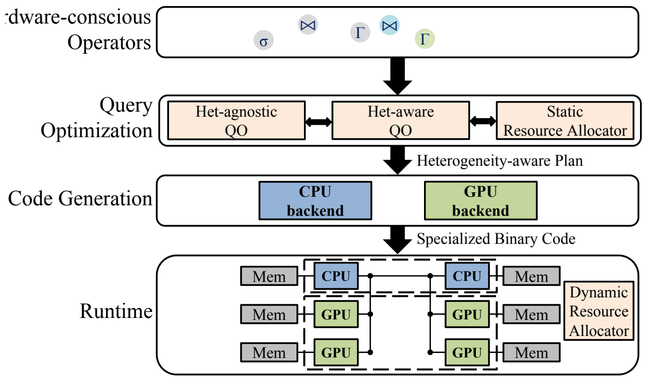

Figure 2: HAPE architecture.

generation provides a unified interface that allows operators to be used on multiple device types. In addition, JIT allows the execution model to be adapted to each device providing enough flexibility for efficient inter-operator execution. HAPE encapsulates the heterogeneity by handling execution and data transfers between devices using the HetExchange operators.

Traits in heterogeneous systems. In a heterogeneous server there are four simple traits [10] that characterize execution: *target devices*, *parallelism*, *data locality* and *data packing*. The first two traits concern the flow of execution, or *control flow*, inside the heterogeneous system. More specifically, for each operation, the first one defines the execution device type, while the second one defines the number of concurrently used devices. The last two traits concern the *data flow* in the system. Data locality is concerned with the distance of the data from their consumer. Transition between different values of any of the control-flow traits requires inter-thread or inter-device task assignment, while increasing data-locality requires data-transfers. All these operations are usually costly, thus it is common practice to amortize their overhead by performing them in the granularity of packets [13]. Unfortunately, decisions often depend on the actual values of each tuple. In such cases we can operate on units of packets only if the property on which the decision depends is common among all the tuples of the packet. Thus, the data packing trait specifies whether the operations operate on tuples or packets and in the case of the later, the properties that are common between all the tuples of each packet. For example, routing packets based on a hash-value implies that for every packet, all its tuples have the same hash. This allows the system to route the packet without actually accessing the contents of the packet.

HAPE architecture. HAPE is composed of three main parts, as shown in Figure 2. The first part is the query optimizer, which is responsible for translating the query into a *heterogeneity-aware physical plan*, a physical plan augmented with information regarding which devices will be used for each part of the tree. By encapsulating conversions of the aforementioned traits in the four HetExchange operators, all relational operators are heterogeneity-oblivious. The heterogeneity-aware plan can explicitly specify the degree of parallelism and target devices of each operator by placing these four operators. Combining the operators with a representation of the plan as a directed acyclic graphs, instead of a tree, permits the plan to use different paths for each device. As a consequence, i) each node of the plan is mapped into a specific device, with the exception of nodes representing a target device conversion, ii) the plan is expressive enough to represent the selection of different algorithms optimized for each target device.

The heterogeneity-aware plan is then broken down into pipelines each targeting a single device. For each pipeline, the code generator produces code optimized for the pipeline's target device through device-specific back-ends, named *device providers*. The generated code is executed on the available devices and is responsible for transferring control and data between the devices. In addition, by coordinating with the scheduler and resource managers, it load balances based on the runtime load.

HAPE benefits. HAPE architecture provides several benefits. First, by encapsulating inter-device operations, HAPE allows relational operators to be heterogeneity-oblivious but also hardware-aware. Relational operators ignore the complexities of remote data, multi-device execution and coordination between devices and focus on using the microarchitecture of their specific target devices as efficiently as possible. At the same time, HAPE provides the methods through four meta-operators to enable co-processing across a mix of CPUs and GPUs. Second, by providing a unified code generation interface, HAPE allows operators to be used for a variety of device types, depending on the needs and the degree of specialization. Third, by embracing control-flow and data-flow operations, it allows load-balancing and data-transfers between the different devices. As a result, HAPE supports query execution both over CPU- and GPU-resident data as well as data scattered over the server's memories. Last but not least, extracting and handling heterogeneity traits through explicit converters makes HAPE compatible with existing query optimizers [4].

HAPE challenges. In order to effectively use the underlying hardware, HAPE has to overcome three challenges. First, it needs efficient operators for single-device execution. As HAPE builds on top of single-device operators, it's overall effectiveness relies on the efficiency of the underlying single-device operators. For CPU query execution there has been debate [29, 3, 2, 5] regarding hardware-oblivious vs hardware-aware algorithms which generally concludes that best option depends on the workload. As HAPE has to support multiple devices, the first challenge is to identify how algorithm specialization and selection differs from CPUs to GPUs.

Second, even if optimal algorithms are used, the inter-operator efficiency can significantly impact performance and in the case of HAPE, the engine should have a common, albeit efficient, execution model to allow hybrid execution. Prior work [7, 21] on CPU query execution has shown the impact of inefficient execution models and tried to minimize them. Recent work [25] has shown that portability can be achieved by expressing the operators in high-level frameworks like OpenCL and/or using vector primitives, but Funke et al. [12] showed that such strategies can incur a high number of passes and thus waste memory (and cache) bandwidth, even when optimized for the GPU-only case. So the second challenge is to identify an execution model efficient both on CPUs and GPUs.

Last but not least, in heterogeneous servers there are multiple devices, cache-coherence is limited, globally shared memory may either not exist or incur high access latencies and inter-device bandwidth is one of the scarcest resources. In order to take advantage of the efficient per-device execution, the engine should be capable of efficiently handling the multiple devices. This requires: 1) that the engine has the necessary mechanisms to efficiently handle transfers and packet routing, 2) the necessary policies and algorithms to decide on the required transfers and routing. So, the third challenge is how can we achieve concurrent multi-device execution in HAPE and how parallel algorithms are mapped in such a system.

HAPE extensibility. While we focus on the multi-CPU multi-GPU case, HAPE is extensible to other accelerators as well. To support a new device type, the engine needs a pair of new device-crossing operators and a device provider. In our prototype the device provider translates the code generation directives to LLVM IR and the generated code contains control flow statements, such as branches and loops. Thus, HAPE is generalizable to such devices.

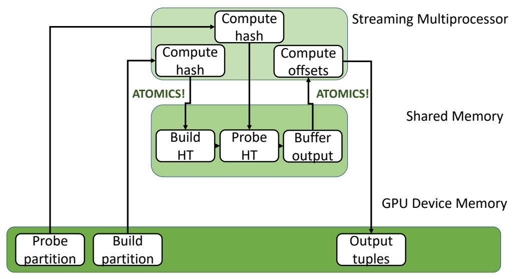

Figure 3: Block diagram for a GPU join over partitioned data.

For devices without control-flow support, but with gather and scatter capabilities, HAPE can be applied by restricting the device providers to such a subset of instructions and allow only the CPU operators to generate more complex code, in order to support routers and allow HAPE to maintain its load balancing capabilities and apply multi-device algorithms.

4. EFFICIENT PARALLEL PROCESSING

4.1 Efficiently parallelizing operators

The abstractions of the Proteus's infrastructure allows the execution engine to be composed of homogeneous subsystems. This property empowers the optimizer to opt for hardware-conscious operators tuned for the specific target device alongside the range of supported hardware oblivious operators. As discussed in Section 2.1, this brings about the potential for significant performance benefits over generic hardware oblivious operators.

Tuning operators for devices. Harnessing the properties of the target device provides the ability to boost performance. Prior-work has employed the range of hardware properties at the level of the operator. One category of optimizations concerns the movement and the access patterns of data with respect to the device's caches, including TLB, and their characteristics. Another category considers the properties and functionality of processing units such as the instruction level parallelism (ILP), affected by pipeline stalls and branch mispredictions, and SIMD instructions for CPUs, and warp-wide execution and shuffles in GPUs. Operator implementations need to incorporate knowledge of properties of the underlying hardware and explore the available opportunities within the design space.

Common design, different specialization. Despite the microarchitectural differences, the exploration of hardware conscious operator designs is not uncorrelated across different devices. Parallels can be drawn between the optimization demands and consequently the design choices across devices. A indicative case is that of hardware conscious hash-joins. In both cases, we adopt partitioning in order to fit the working sets in cache and reduce expensive random accesses. Additionally, the partitioning fanout is restricted by the TLB size on the CPU side and the size of the cache that contains write offsets on the GPU side. The end result is a multi-pass partitioned hash-join design.

In GPUs, it is possible to do some further optimizations. Random accesses to L1 waste bandwidth as a whole cache line has to be fetched per access. By contrast, to make the probing step GPU-friendly, we load the smaller partition to the scratchpad, build

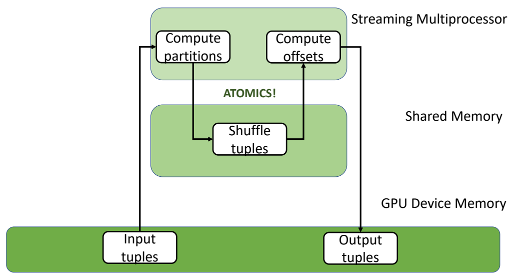

Figure 4: Block diagram for a GPU partitioning pass.

the hash-table using atomics and probe with the tuples of the corresponding partition. The scratchpad is organized into 32 banks and is capable of serving a different word from each bank per (warp-)request, independently of its location in the bank. Thus, the scratchpad only penalizes access to the same bank, but does not waste bandwidth by over-fetching. We show a block diagram of the build & probe sequence within the memory hierarchy in Figure 3.

As the scratchpad is of limited size, similar to the size of L1, the produced partitions of the two inputs should be small enough to fit in it. Therefore, the number of partitions produced should be sufficiently high. In the CPU case, the partitioning is optimized with the goal to reduce the TLB misses and improve the cache locality of the output. By contrast, for GPUs we aim to reduce the sparsity of the stores. To achieve this, we shuffle a chunk of the input at-the-time in the scratchpad and reorganize it in such a way that elements belonging to the same output partition are located in consecutive scratchpad entries. Then, we scan the scratchpad and write each tuple to its corresponding output partition. By controlling the number of output partitions, we control the average number of elements mapped to each partition for each step. The reordering gathers elements resulting to the same partition together and thus increases the coalescing of the stores. Coalescing allows for better utilization of the high memory bandwidth of GPUs and improves the throughput. Of course, this means that we prefer few output partitions and multiple passes over the data might be required to achieve scratchpad-resident co-partitions. Contrary to [28]'s GPU hash-join, in each partition pass we scan the data once and write them to linked list of buffers managed with atomics. This technique avoids performing an extra scan to determine the output offsets. We illustrate the block diagram for the steps of a partitioning pass within the memory hierarchy in Figure 4.

The GPU hardware conscious operator considers the specific memory hierarchy of the GPUs and is tuned for it. However, the skeleton of the algorithm remains the same for both CPUs and GPUs. We can observe that the design of the hardware conscious operators has two main components: the algorithmic skeleton and the hardware-specific finer-grained building blocks such as caching the hash table in the scratchpad. The same parallels exist for other hash-based relational and set operators. Of course, this does not entail that the specific implementation is the optimal choice for a specific query plan (i.e. a sort-merge join can be preferable at other times). Nevertheless, this observation provides an insight for designing operators by decoupling the core algorithm from the exposition of specific hardware properties and mapping the design spaces of different devices.

4.2 Efficiently parallelizing query plans

In order to achieve efficient inter-operator CPU and GPU query execution, we use code generation to produce efficient code for each target device and we parallelize the execution to multiple homogeneous devices by scheduling execution of the generated code as well as any necessary data transfers. In Section 5 we discuss how these techniques extend concurrent execution to multiple heterogeneous devices.

Code generation. We use code generation to achieve two goals, i) a unified interface for operators to target multiple devices and ii) enough flexibility to provide a hardware optimized implementation.

We provide the implementation of the code generation interface with a different back-end per device. Each back-end is responsible for producing code tailored to the underlying device. Starting from the lower level, back-ends are responsible for translating code generation directives received by the operator to the instruction set of their target device. In a higher level, they specialize common functions, like worker-scoped atomics, reductions and synchronizations to the underlying device. For example, a back-end for single-threaded CPU execution would optimize-out worker-scoped atomics to simple load-apply-store operations.

Homogeneous inter-device parallelism. In order to achieve inter-device parallelism over a set of homogeneous device, we extend the traditional Exchange [13]. Similar to the Exchange, we instantiate both the producer and consumer code on multiple, possibly different, devices to achieve the desired input and output degrees of parallelism. In contrast with the traditional Exchange, we separate *control flow*, transfers between producers and consumers, from *data flow*, transferring data over the interconnects. Treating each of them separately, allows us to take data-dependent decisions to transfer control without access to the data at the point of decision as well as perform load balancing.

As control flow operations are inherently more CPU-friendly than GPU we propagate task assignment and load balancing to the CPU and perform them through a CPU-side operator, HetExchange's *router*. This operators is the converter for the parallelism trait and receives from producers tasks that routes to consumers, based on policies. Our implementation has load-aware, locality-aware and hash-based policies. We push down to the producers the responsibility of packing data such that the router can take the routing decision in a packet granularity, without accessing the data and only based on packet metadata.

Depending on the routing policy, a packet may be routed to a consumer that does not have access to its content. To handle such cases, we represent data transfers as an operator and place them on the plan. In addition, a variant of the same operator takes into consideration the memory topology in order to perform broadcasts with minimal number of copies. By taking into consideration the memory topology, this performs packet multi-casts and copies, in order to minimize the data copies during broadcasts. In addition, decoupling the data transfers from the relational operators allows our design to operate over both initially CPU-resident and GPU-resident datasets as well as datasets that are distributed over all the CPU and GPU memory nodes.

5. EFFICIENT CO-PROCESSING

Efficient multi-CPU multi-GPU query processing under a single engine allows executing the query with the more appropriate device. Nevertheless, the rest of the devices may still be underutilized. The rest of this section expands our design to concurrently use all the available heterogeneous processing power.

Similarly to the homogeneous case [13], there are three ways to parallelize a query over heterogeneous devices. First, the engine can vertically partition the query plan and execute each part on the most appropriate type of devices, pipelining execution between heterogeneous devices. Second, different subtrees of the query plan can be distributed to different devices, thus partitioning the plan horizontally. Third, we can design efficient operators, the execution of which spans across multiple heterogeneous devices.

Vertical co-processing. We achieve pipelined execution across devices by exposing the fact that the two vertical partitions of the plan are independent which allows us to independently select the more appropriate algorithms for each part as well as generate efficient code for the target device of each part. We encode the transition between device targets using HetExchange's *device crossing* that is responsible for changing the back-ends used during the code generation and handles the transition of execution between different devices, effectively hiding from the other operators that their input is possibly received from another device, both during code generation and execution time.

Horizontal co-processing. The design supports horizontal parallelism across multiple heterogeneous devices by allowing routers to have multiple distinct parents. By this relaxation, each of its parent can transfer relational operators into different types of devices. For example, in order to split an aggregation across 48 CPU cores and 2 GPUs, the plan has a router operator that runs on the CPU with two parents. The first parent is an aggregation while the second one is a CPU-to-GPU device crossing operator followed by an aggregation. For this example, the first parent would be instantiated 48 times, resulting in the parallel execution of the aggregation to the CPU cores. The device crossing operator and its aggregation would be instantiated 2 times by the router, for each instance to transfer the execution to its GPU and use it for the aggregation. The router does not differentiate between parents based on where they execute the main computations, the device crossing will handle that. Data partitioning required for the horizontal heterogeneous parallelism is handled as described in Section 4.2.

Intra-operator co-processing. In order to provide efficient algorithms for co-processing, it is possible to combine without modification algorithms tailored to each device via data partitioning and scheduling policies, reducing this way the design effort. Continuing on the radix-join algorithm of Section 4.1, Sioulas et al. [31] propose to use the CPU in order to perform locally to the input a first partitioning step that allows us to perform the join with a single pass over the slow interconnect. The two inputs of the join are co-partitioned in their initial location in such a way that each co-partition can fit in GPU memory. Then, for each pair, we transfer it to the GPU and execute the more fine grained partitioning steps and the probing as described in Section 4.1. The CPU-side partitioning requires a smaller number of partitions compared to the the full partitioning required for the radix join and thus can be optimized to achieve very high throughput even for datasets in the order of tens of gigabytes. Then, by controlling the number of partitions, we fit each co-partition in the GPU memory and thus a single pass over the interconnect is required, as long as not single key has o many tuples that do not fit in GPU memory. Parallelizing this algorithm across multiple CPU cores or GPUs is achieved as described in the previous section.

The task of selecting the above server-wide algorithm is propagated to the query optimizer. The query optimizer places an initial CPU-side partitioning operator on each of the two inputs. These operators are followed by a zip operator that matches the corresponding partitions from each side into co-partitions and pushes them to the next operator. The zip is followed by a split operator which drives each of the two partitions to a (different) sequence of a mem-move, a CPU-to-GPU and another partitioning operator. The co-partitions produced by the latter are then zipped once more, unpacked and propagated to the actual in-GPU join operator.

Based on the above plan, the query optimizer can also produce

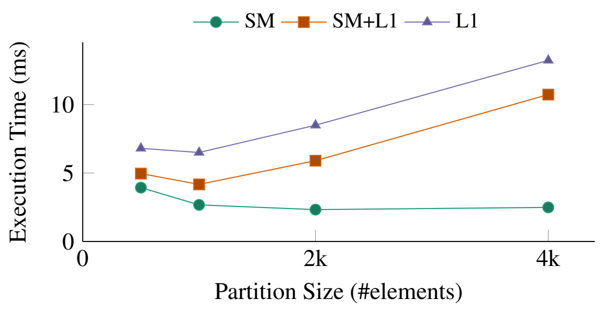

Figure 5: Scratchpad (SM) vs L1 during GPU radix's probing phase

other more complex ones using its optimization rules. For example, instead of sending all the co-partitions to a single GPU, it can add a router to send some co-partitions into a second GPU, or even keep some of them for joining on the CPU-side.

6. EVALUATION

In this section, we present an evaluation of the performance for the system described above.

6.1 Experimental Setup

The following experiments run on a machine provisioned with two 12-core Intel Xeon E5–2650L v3 running at 1.8 GHz, with 64KB of L1 and 256KB of L2 cache memory per core, 30MB of shared L3 cache and 256 GB of main memory. Also, the machine is equipped with two NVidia GeForce GTX 1080 GPUs each with 8 GB of local memory and connected one of the two CPU sockets, through a dedicated PCIe 3 x16 interconnect. We compare our implementation with two state-of-the-art commercial systems, DBMS C and DBMS G. DBMS C is a CPU-based columnar DBMS that is based on MonetDB/X100 [7], uses SIMD vector-at-a-time execution and supports multi-CPU execution. DBMS G is a GPU-based DBMS that supports multi-GPU execution and uses just-in-time code generation for the in-GPU kernels.

6.2 Hardware-conscious Join

This section compares the GPU partitioned hash-join against other algorithms and implementations. The first two experiments use two equally-sized tables, each with two 4-byte columns: a key and a payload. We measure the performance of an equi-join over the key columns that is followed by a sum/count aggregation over each payload column. Both tables have exactly the same keys and thus the output has as many tuples as any of the inputs.

Figure 5 assesses the importance of using the GPU scratchpad for the build and probe phase of the join, rather than following the CPU conversion of using the L1. For this experiment, we use 32 million tuples for each table, load them in GPU memory and measure the execution time of the in-GPU partitioned join for different numbers of partitions. This maintains the same input size but changes the number of elements per partition. To filter-out the effects of handling over-sized partitions and focus on the impact of the memory, we select the keys to produce all the partitions with exactly the same size. Each co-partition is assigned to a GPU block of threads and thus this defines the memory requirements for the intermediate structures per block. We compare three variants: L1, which optimistically stores all the corresponding data in L1, SM that stores all the data of the build table in the scratchpad and SM+L1 that stores the offsets of the heads of the hash table chains in scratchpad the rest in L1.

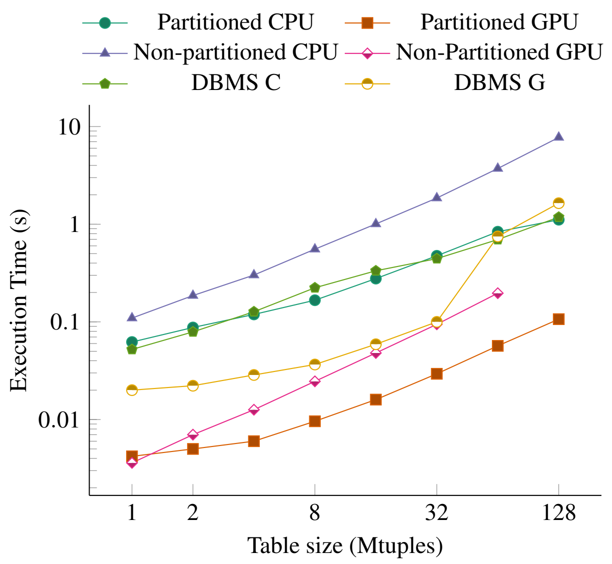

Figure 6: Comparison of parallel CPU and (single) GPU joins

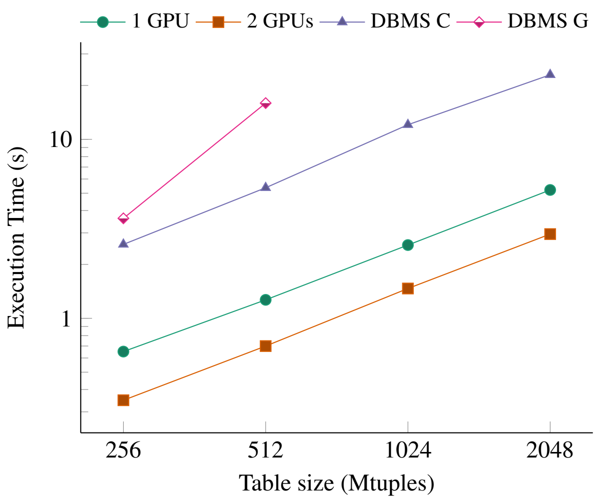

Figure 7: Comparison of join co-processing using 1 and 2 GPUs

The more we rely on the scratchpad to store join's intermediate structures, the better the performance, as the scratchpad, in contrast with L1, is not overfetching. In addition, L1 is affected by the scanning of the co-partitions, which causes cache pollution and decreases the hit rate proportionally to the input size. In contrast, the scratchpad is software managed and thus it's not affected by the same problem. As a result, the performance of the scratchpad is almost constant while the performance of L1-based solutions decreases as the partitions increase. SM+L1 has the advantage that the first probe is in the SM, but it is also affected by the drawbacks of the L1-based solution. It also worths mentioning that SM+L1 has an increased capacity, compared to the other two solutions. The small performance degradation from 1024 to 512 elements per partition is due to hardware underutilization: the small partition size reduces the opportunities for useful overlapping.

Figure 6 focuses on the in-CPU/-GPU performance of partitioned and non-partitioned CPU and GPU joins implemented in our system, as well as with the join implementations of DBMS C and DBMS G. In this experiment we plot the execution time for varying table sizes from 1 million to 128 million tuples, at which point the datasets stop fitting in the GPU memory. In each case, the data are pre-loaded to the local memory of the compute unit. Due to it's improved hardware utilizations, the GPU hardware-conscious algorithm outperforms all the alternatives, with an over 3x speed-up against the non-partitioned variant for the highest supported in-GPU size and over an order of magnitude for the 128 million tuple datasets against the other implementations.

6.3 Operator-level co-processing

The next experiments evaluates the join co-processing technique designed for scaling up the join to cases when the GPU-memory is insufficient or storing the inputs tables and intermediate join structures. In that purpose, we scale to datasets bigger than the ones in Figure 6, from 256 million tuples to 2 billion tuples and operate over CPU-resident data. Figure 7 shows the execution time of the co-processing technique for the case of 1 and 2-GPUs compared against the joins of the two commercial systems. DMBS G is not designed for out-of-GPU datasets, and thus performs purely even after 512 million tuples. DBMS G scales linearly with the number of keys but despite the fact that it has access to the data with the DRAM bandwidth, the random accesses force CPU implementations to suffer either from high latencies, or reduce the latencies at the expense of multiple passes, which causes the DMBS C to achieve a throughput significantly lower than the PCIe throughput. In contrast, the co-processing approach is achieving the best from both worlds. It partitions the data in the CPU-side were it has the DRAM bandwidth in its availability for scanning the data. On top of that, as it requires a relative low-fanout, the partition materialization can also take advantage of the high DRAM bandwidth. The partitioning allows for a single-pass over the slow PCIe and on the GPU-side, the 280GBps memory bandwidth of each GPU in combination with the optimizations to use the scratchpads allow for a partition-and-join throughput also higher than the PCIe. As a result, in the single-GPU co-processing the join in bottlenecked by the PCIe which is higher that the CPU-only join throughput. In addition, adding an extra GPU, on a dedicated PCIe bus, almost doubles (1.7x) the total throughput as the GPU-side join throughput is also doubled, while the near-DRAM low-fanout CPU-side partitioning can sustain providing for the two PCIes. Overall, Proteus achieves 12.5x and 4.4x speedup over DBMS G and DBMS C, respectively, for the largest dataset size each of them supports.

6.4 Query Plan-level co-processing

The rest of the experiments focus on evaluating the end-to-end performance of presented query engine and more specifically evaluate its efficiency on achieving state-of-the-art performance in each device and in hybrid mode, achieving the sum of the throughputs of each individual device. We use four TPC-H queries, at scale factor 100, to evaluate our system: Q1, Q6 which are simple aggregations and thus stress the interconnect and memory bandwidth utilization of the system, and Q5, Q9 that are join-heavy. As we currently have no support for LIKE conditions, we run Q9 without the LIKE condition and the join to the corresponding table. We use a binary columnar format for the inputs, which for our system is translated to 15-27GB working sets per query. Taking into consideration intermediate data structures and space for buffer management, non of this queries fits in the memory of the two GPUs. Thus, for all the experiments and systems the data are CPU-resident. For each experiment we warm up the systems and allow them to load the data in-memory prior to the actual measurements.

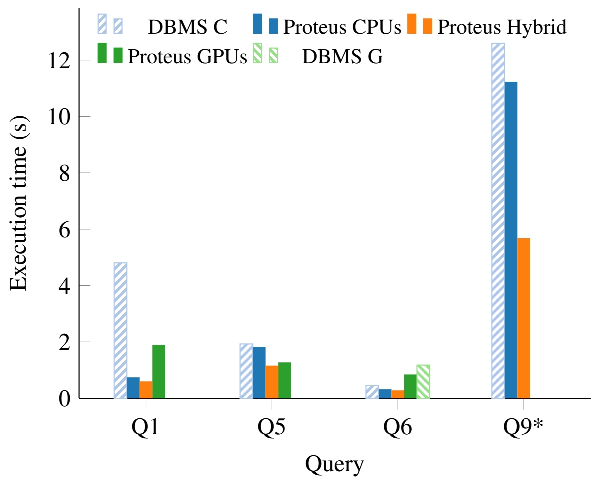

Figure 8: CPU-, GPU-only and Hybrid performance on TPC-H

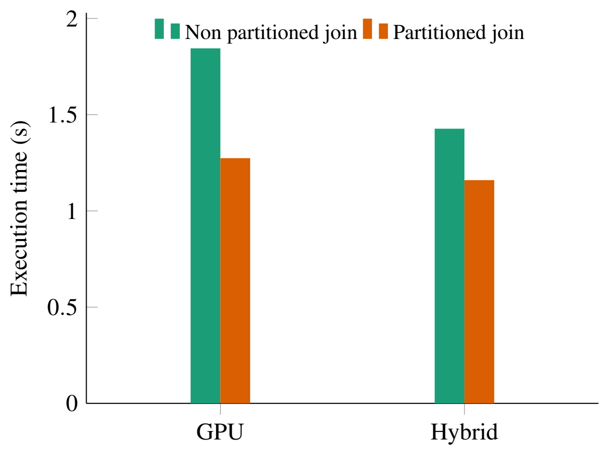

Figure 9: Partitioned vs Non-Partitioned-based join on TPC-H Q5

Figure 8 plots the execution times for the commercial systems and for three configurations of Proteus: CPU-only, GPU-only and hybrid (multi-CPU multi-GPU) execution. The performance of Proteus CPU is comparable DBMS C, with the exception of Q1. Q1 has multiple aggregates and thus DBMS C has a higher overhead due to the multiple in-L1 passes required by its vector-at-a-time processing. In contrast the JIT code generation of Proteus CPU avoid that. DBMS G is optimized for star-schema based queries and in-GPU processing and thus it was unable to run on 3 out of queries.

For the case of single device execution, we see that the relative performance depends on the query category. For the scan-bound queries, CPU-only execution demonstrates significantly better performance than GPU-only execution and is more than 2.65 times faster for both queries. The CPUs only access local DRAM and sustain a throughput higher than the combined bandwidth of the interconnects over which data are transfered to the GPUs. However, for the join-intensive Q5, GPU-only execution attains higher performance and achieves a speedup of 1.4x despite the data transfers over PCIe. This upfront overhead is amortized as the heavy processing involved benefits from the hardware capabilities and the fine-tuned algorithm on the GPU, while the CPU-side join suffers from high latencies and/or multiple passes. Q9 is producing intermediate results that scale the hash-table requirements further than the available memory on the GPUs and thus none of the GPU-only systems is able to execute it.

Even though the choice between CPU-only and GPU-only execution is case-by-case, in all four experiments the multi-CPU multi-GPU hybrid configuration outperforms both in all scenarios. The execution mode is most efficient for Q1 and Q6 queries, as it can achieve 89% and 82% of the sum of throughput of that is achieved by the CPU-only configuration plus the GPU-only configuration. For Q5 the hybrid configuration achieves an 64% of the aggregate throughput, due to the overhead of shuffling data for the joins. Additionally, hybrid execution allows for co-processing at the operator level which is the cornerstone for evaluating Q9. The co-processing join technique presented earlier is combined with in-GPU joins to provide a speedup of 2x over the CPU version. This result shows the practical value of the technique.

Figure 9 depicts the execution time for GPU-only and multi-CPU multi-GPU variants for query Q5, with a partitioned join as a representative example of a hardware-conscious join and a non-partitioned join as the representative for the hardware-oblivious joins, for the heavy joins on the GPU-side of the plan, in order to outline the impact of optimized operators within the query plan. The plans that contain the partitioned joins have a lower execution time, with 1.44x and 1.23x speedups for GPU and hybrid execution respectively. The efficient hardware-optimized operator is able to mitigate the join bottleneck and increase performance and showcases the importance of hardware-conscious processing, providing a strong case for HAPE design.

7. CONCLUSIONS

In conclusion, we presented HAPE, a design for analytical query engines that achieves efficient query execution over heterogeneous hardware. We showed that heterogeneous execution can be decomposed into a two dimension problem, achieving efficient inter-device execution and developing the mechanism and abstractions necessary for transferring control and data between devices. Efficient inter-device execution can be achieved by efficiently combining hardware-tailored operators.

8. ACKNOWLEDGMENTS

This project has received funding from European Union Seventh Framework Programme, 2013 - ERC-2013-CoG, grant agreement number 617508, ViDa and H2020 - UE Framework Programme for Research & Innovation (2014-2020), 2017 - ERC-2017-PoC, grant agreement number 768910, ViDaR.

References

[1] R. Appuswamy et al. The case for heterogeneous htap. In *CIDR*, 2017.

[2] C. Balkesen et al. Main-memory hash joins on multi-core cpus: tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

[3] C. Balkesen et al. Multi-core, main-memory joins: sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, Sept. 2013. ISSN: 2150-8097. DOI: 10.14778/2732219.2732227. URL: http://dx.doi.org/10.14778/2732219.2732227.

[4] E. Begoli et al. Apache calcite: a foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*. ACM, 2018.

[5] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 37–48, Athens, Greece, 2011. ISBN: 978-1-4503-0661-4.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 1999. ISBN: 1-55860-615-7. URL: http://dl.acm.org/citation.cfm?id=645925.671364.

[7] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[8] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*, pages 1891–1906, 2016.

[9] T. Chen et al. Tvm: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[10] P. Chrysogelos et al. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB*, 2019.

[11] T. P. P. Council. Tpc-h benchmark specification. *Published at http://www. tcp. org/hspec. html*, 21:592–603, 2008.

[12] H. Funke et al. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1603–1618. ACM, 2018.

[13] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[14] B. He et al. Relational Query Coprocessing on Graphics Processors. *TODS*, 34(4):21:1–21:39, 2009.

[15] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013. ISSN: 2150-8097. DOI: 10.14778/2536206.2536216. URL: http://dx.doi.org/10.14778/2536206.2536216.

[16] M. Heimel et al. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[17] H. Inoue et al. Aa-sort: a new parallel sorting algorithm for multi-core simd processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 189–198, Sept. 2007. DOI: 10.1109/PACT.2007.4336211.

[18] T. Kaldewey et al. GPU join processing revisited. In *DaMoN*, 2012.

[19] T. Karnagel et al. Big data causing big (tlb) problems: taming random memory accesses on the gpu. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, page 6. ACM, 2017.

[20] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12):972–983, 2016.

[21] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[22] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[23] MapD. https://www.mapd.com/.

[24] J. Paul, J. He, and B. He. GPL: A GPU-based Pipelined Query Processing Engine. In *SIGMOD*, pages 1935–1950, 2016.

[25] H. Pirk et al. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.

[26] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, Melbourne, Victoria, Australia. ACM, 2015. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2747645. URL: http://doi.acm.org/10.1145/2723372.2747645.

[27] G. Psaropoulos et al. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment*, 11(2):230–242, 2017.

[28] R. Rui and Y. Tu. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *SSDBM*, 17:1–17:12, 2017.

[29] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1961–1976, San Francisco, California, USA. ACM, 2016. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882917. URL: http://doi.acm.org/10.1145/2882903.2882917.

[30] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 1994. ISBN: 1-55860-153-8. URL: http://dl.acm.org/citation.cfm?id=645920.758363.

[31] P. Sioulas et al. Hardware-conscious Joins on GPUs. In *ICDE*, 2019.

[32] E. Stehle and H.-A. Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 417–432, Chicago, Illinois, USA. ACM, 2017. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064043. URL: http://doi.acm.org/10.1145/3035918.3064043.

[33] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *PVLDB*, 6(10):817–828, 2013.

[34] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, Madison, Wisconsin. ACM, 2002. ISBN: 1-58113-497-5. DOI: 10.1145/564691.564709. URL: http://doi.acm.org/10.1145/564691.564709.