HetExchange: Encapsulating heterogeneous CPU–GPU parallelism in JIT compiled engines

Periklis Chrysogelos[†] Manos Karpathiotakis[‡] Raja Appuswamy[§] Anastasia Ailamaki[†¶]

[†]EPFL, [¶]RAW Labs SA
{firstname}.{lastname}@epfl.ch

[‡]Facebook
manos@fb.com

[§]EURECOM
raja.appuswamy@eurecom.fr

ABSTRACT

Modern server hardware is increasingly heterogeneous as hardware accelerators, such as GPUs, are used together with multicore CPUs to meet the computational demands of modern data analytics workloads.Unfortunately, query parallelization techniques used by analytical database engines are designed for homogeneous multicore servers, where query plans are parallelized across CPUs to process data stored in cache coherent shared memory. Thus, these techniques are unable to fully exploit available heterogeneous hardware, where one needs to exploit task-parallelism of CPUs and data-parallelism of GPUs for processing data stored in a deep, non-cache-coherent memory hierarchy with widely varying access latencies and bandwidth.

In this paper, we introduce HetExchange–a parallel query execution framework that encapsulates the heterogeneous parallelism of modern multi-CPU–multi-GPU servers and enables the parallelization of (pre-)existing sequential relational operators. In contrast to the interpreted nature of traditional Exchange, HetExchange is designed to be used in conjunction with JIT compiled engines in order to allow a tight integration with the proposed operators and generation of efficient code for heterogeneous hardware. We validate the applicability and efficiency of our design by building a prototype that can operate over both CPUs and GPUs, and enables its operators to be parallelism- and data-location-agnostic. In doing so, we show that efficiently exploiting CPU–GPU parallelism can provide 2.8x and 6.4x improvement in performance than state-of-the-art CPU-based and GPU-based DBMS.

PVLDB Reference Format:
Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: Encapsulating heterogeneous CPU–GPU parallelism in JIT compiled engines. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

The past few years have witnessed the transformation of Graphics Processing Units (GPU) from niche processors used in the gaming and visualization industry to more general-purpose accelerators used in various analytical, data-intensive applications. Modern General-Purpose GPUs provide massive parallelism as they are equipped with thousands of cores organized in a Single-Instruction-Multiple-Thread execution model. Due to the use of tightly integrated high-bandwidth memory, GPUs also provide an order of magnitude faster access to local memory than CPUs. Thus, GPUs are being used in several deployment scenarios ranging from supercomputers used for HPC applications, through on-premise enterprise servers equipped with multiple GPUs, to Platform-as-a-service offerings that provide GPU-accelerated virtual machines. As a result, database engines are being increasingly deployed on heterogeneous hardware equipped with both CPUs and GPUs. Unfortunately, even state-of-the-art database systems fall short in fully exploiting available heterogeneous processing capacity.

Traditionally, analytical DBMS have operated solely over CPUs. For several decades, the *Exchange* infrastructure introduced by the Volcano interpreted query execution framework has been the standard way of parallelizing query execution. By using a family of Exchange operators that can be injected into the query plan to connect unmodified sequential producer and consumer operators using queues, Volcano made it possible to achieve horizontal, vertical, and bushy parallelism both within and across servers. However, the interpretation overhead of traditional Volcano-style query execution has been shown to be a performance bottleneck in modern in-memory database engines. Thus, over the past few years, JIT compilation has regained popularity as a way of avoiding such overheads, and new approaches for parallelizing query execution based on JIT compilation that expose, rather than encapsulate, operators to parallelism have been adopted by in-memory analytical engines. Unfortunately, these new techniques fundamentally rely on homogeneous task parallelism of CPUs to parallelize single-threaded, compiled pipeline tasks across multiple CPUs, and cache-coherent shared memory provided by multicore CPUs for performing atomic operations on shared data structures. Thus, they are not applicable in the GPU context due to the differences in the type of parallism across CPUs and GPUs, and a lack of system-wide cache coherence for implementing global atomics.

Given these issues, a number of recent research and industrial DBMS have explored design alternatives for parallelizing analytical queries on GPUs [1, 5, 13, 14, 16, 27, 34, 28, 2], yet they apply numerous simplifying assumptions. First, several GPU DBMS typically support execution over a single GPU instead of multiple ones [9, 5, 13, 14, 27, 34, 2] as they lack the abstractions to express cross-GPU execution. Second, they typically assume that data is partitioned and pre-loaded in GPU device memory [27] in order to avoid the cost of data transfers during query execution via the PCI Express interconnect. Such assumptions severely limit the range of query plans that can be supported [1, 16]. Third, most GPU engines cannot execute queries on multicore CPUs. Thus, these engines leave substantial amount of processing capacity under utilized when deployed on modern heterogeneous servers. Fourth,

Proceedings of the VLDB Endowment, Vol. 12, No. xxx

DOI: https://doi.org/TBD

heterogeneity-aware analytical engines parallelize queries on CPUs *or* GPUs [14, 28] and not across CPUs *and* GPUs. Thus, state-of-the-art lacks a single, unifying mechanism that can combine the efficiency of JIT compilation with the ability to encapsulate parallelism like Exchange across heterogeneous processors.

This paper presents *HetExchange*–a framework to encapsulate the heterogeneous parallelism in modern servers to enable analytical query execution across multiple CPUs and GPUs. Similar to traditional Exchange [10], HetExchange encapsulates parallelism and provides a uniform interface to connect producers and consumers in a pipelined plan together with the memory infrastructure. However, unlike traditional Exchange, which dealt only with homogeneous parallelism across CPUs, HetExchange encapsulates heterogeneous parallelism across CPUs and GPUs. Additionally, unlike Exchange, which connects individual operators in an interpreted execution environment, HetExchange connects subpipelines in a JIT compiled execution environment. Thus, HetExchange provides a framework that can be used by JIT compiled engines to parallelize sequential, single-threaded code on multiple CPUs and single-GPU kernels across multiple GPUs, or even a single query plan across both CPUs and GPUs in a coprocessing fashion. In doing so, HetExchange shares the benefits of the two popular parallelization techniques without the disadvantages. By encapsulating heterogeneity, HetExchange proposes a single abstraction that can be used to encapsulate heterogeneous parallelism without making assumptions about hardware characteristics, like the availability of globally cache-coherent shared memory. Furthermore, by integrating tightly with JIT compilation, HetExchange eliminates the overhead of interpretation and enables register pipelining optimizations that have been pioneered by JIT engines.

Contributions. The contributions of this work are the following:

- We introduce HetExchange–a novel parallel query execution framework that encapsulates heterogeneous parallelism to enable query plan deployment across i) CPUs, ii) GPUs and iii) mix of CPUs and GPUs
- We detail the design of a HetExchange augmented JIT compiled engine, and present an evaluation of our prototype engine against state-of-the-art CPU-based and GPU-based engines; our prototype is up to 1.5x and 5x faster when restricted to the same compute units and up to 5.1x and 11.4x faster, respectively, when utilizing the whole machine, while at the same time achieves linear scalability.

2. BACKGROUND AND RELATED WORK

In this section, we provide an overview of the hardware setup that is typical in today's heterogeneous compute servers and summarize related work on parallelizing query execution in order to set the context for our work.

2.1 Heterogeneous parallelism in modern servers

Modern servers incorporate numerous accelerator devices – typically multiple GPUs, connected to each CPU socket via a PCIe (3.0) interconnect. When more than one GPU device is attached to CPU socket, the server can utilize a PCIe switch to increase the number of PCIe connections per socket. Overall, the modern server is becoming increasingly heterogeneous: It is equipped with diverse processors, organized in non-uniform memory access topologies.

CPUs experience additional memory access latency overhead when a core accesses memory that is attached to a socket different than the one that the core is attached on – a phenomenon dubbed NUMA (non-uniform memory access) [20]. Introducing GPUs as additional processors exacerbates NUMA effects. When a GPU accesses CPU memory, it transfers data through the PCIe interconnect, whose bandwidth ($\sim$16GB/s for PCIe 3.0) is limited compared to the bandwidth of a CPU's local DRAM ($\sim$ 80GB/s) and to the bandwidth of a GPU's device memory (up to 900GB/s). In addition, if the server relies on PCIe switches to connect multiple GPUs to a CPU socket, then the per-switch GPUs have to share the PCIe bandwidth if both of them trigger PCIe traffic.

2.2 Parallel query execution on CPUs

Volcano and Exchange. When a query is posed in a database system, it is processed by a query planner / optimizer, resulting in an algebraic plan. This plan, expressed in the form of a tree, was traditionally interpreted using the Volcano iterator model [10]. Every operator of the plan exposes a general API, consisting of *open()*, *next()* and *close()* functions. When an operator's *next()* method is called, a request for a new tuple is sent to the operator's children.

Exchange operator introduced in Volcano has been the standard approach for parallelizing a query plan. The Exchange operator encapsulates all three different types of parallelism (horizontal, vertical, and bushy) by exposing the same (iterator) interface as other operators in an interpreted query plan. Inserting an Exchange operator in a query plan splits it into two parts, with the sub-plan above the Exchange becoming the consumer and the sub-plan below being its producer. The Exchange operates as an asynchronous queue between the producer and the consumer. The producer inserts its results into the queue, while the consumer removes them and processes them. Both the producer and the consumer are not aware of the queue and they interface to the Exchange operator with the same interface as with any other operator. As both producer and consumer can execute in parallel on different processors, Exchange enables vertical parallelism. In addition, the Exchange controls the degree of parallelism of the consumer and producer by spawning multiple instances of them and routing packets between the different instances, introducing this way horizontal parallelism. Producers' results are either routed based on a policy to exactly one consumer or broadcasted to all of them. Lastly, introducing Exchange operators in both sides of a join creates bushy parallelism.

JIT compilation and exposing parallelism. Although Exchange makes it possible to parallelize sequential, single-threaded operator implementations without any code changes, it has certain drawbacks that limit its applicability as the mechanism of choice for parallelizing query execution in the modern in-memory data processing context. Interpreted query execution penalizes performance as the *next()* function is called for every tuple, resulting in frequent branch misprediction and poor code locality [26, 18].

State-of-the-art in-memory analytical engines avoid such interpretation overhead by eschewing intepreted execution in favor of JIT compilation. JIT-based in-memory database engines split the query plan into non-blocking pipelines and use a compiler framework to translate a sequence of operators into straight-line code that loops over data one tuple at a time. Thus, these systems enable register-pipelining, as a collection of non-blocking operators can be applied in one shot to a tuple stored in CPU registers. CPU-based JIT compilation techniques also do not use the traditional Exchange-based parallelism where operators, other than Exchange, are essentially sequential in nature. Instead, the typical approach, as exemplified by Hyper's morsel-driven parallelism [21] is to compile and generate parallelization-aware operators by using atomic instructions in generated code for synchronizing access to shared data structures. Such code can then be executed in a task-parallel manner across multiple CPUs by using a thread pool.

CPU parallelism in heterogeneous servers. Unfortunately, the aforementioned approaches for parallelism can not be used in modern heterogeneous servers to parallelize queries across CPUs and

GPUs. The traditional Exchange was not designed to work in heterogeneous parallel processing environments, as pipelining across different processors requires the asynchronous queues of the operator to be placed such that they can be efficiently accessed by all processors. Both Exchange and JIT compilation require system-wide cache coherence as they rely on atomic operations for synchronizing access to producer–consumer queues, or other shared data structures like the hashtable used during the build phase. While these assumptions hold in a homogeneous multicore CPU server, the same cannot be said about heterogeneous servers with CPUs and GPUs due to a lack of global cache coherence or system-wide shared memory.

Further, unlike CPUs, executing an operator on the GPU requires moving the input data to GPU memory, launching a kernel to process the input, and potentially moving out the output data. As moving data is an expensive operation, it is important to move enough data so that the benefit gained from processing data on the GPU outweights data movement cost. Similarly, since kernel launches are expensive and slow, it is also important to minimize the number of kernel launches. Unlike a CPU-based JIT compiler, which has to generate executable code for just one processor, GPU-based JIT compiler should generate both kernels that are executed on the GPU and host code that runs on CPUs that invokes GPU kernels. Thus, modern servers with heterogeneous CPU–GPU parallelism require rethinking traditional query execution strategies.

2.3 Parallel query execution on GPUs

Operator-at-a-time execution. The inability of the commodity CPU to achieve unconditional scalability has led numerous research and industrial efforts that utilize GPU co-processors for the acceleration of analytical database workloads [5, 13, 14, 16, 27, 34, 28]. Most GPU-powered DBMS operate as follows: The DBMS expresses the query plan as a sequence of (micro-)operators [5, 13, 14, 16, 34], and then translates each operator into a *kernel* – a data-parallel function. The DBMS then executes the kernels, one after the other, on a GPU, fully materializing intermediate results in order to provide them as input to the next kernel.

Initially, such "operator-at-a-time" GPU DBMS [5, 14] required every kernel to have its input available at operator invocation time, and thus complicated the overlap of (GPU) computation and (CPU-to-GPU) data transfer. Subsequent systems thus introduce the following optimizations. First, they overlap data transfer with computation to mask the data transfer cost as much as possible. For example, GPUDB [34] uses *Universal Virtual Addressing (UVA)*, an NVIDIA CUDA feature that allows a GPU to directly access memory of the CPU side, while the authors of [29] use CUDA memory copies and CUDA streams for a similar purpose. The authors of [31] propose CPU-GPU co-processing to accelerate sorting tasks; their approach parallelizes the production of sorted runs, which it interleaves with data transfers to and from the GPU. Second, modern GPU DBMS have followed the MonetDB/X100 [4] paradigm to reduce the materialization overhead [26] between kernel invocations; every kernel operates over a subset (i.e., a vector) of the input, and likewise produces a vector as its output. The intermediate result vector thus fits on GPU memory for the next kernel to read, and the DBMS avoids unnecessary data transfers of intermediate results to the CPU host. Still, result materialization – even if it involves vectors – between kernel invocations is wasteful in terms of memory bandwidth [9]; the GPU DBMS has to flush GPU registers and shared memory between kernel invocations, thus hurting locality. In addition, the vector-at-a-time paradigm requires multiple passes, thus further wasting (GPU) memory bandwidth.

Pipelined GPU execution. An alternative to vector-at-a-time processing is performing as much work as possible over data that already resides in GPU registers / shared memory. Such *pipelined* query execution typically reduces the number of kernels per query plan. GPL [27] pipelines operators by having each one of them running on a separate kernel, and having the kernels communicating and transferring data through *OpenCL 2.0 pipes* [12]. HAWK [7] is a query compiler that generates OpenCL code; the produced code can execute on a variety of parallel processors, such as CPUs and GPUs; the focus of HAWK is on having the generated code execute entirely on a single of these platforms (i.e., on a CPU or a GPU). HorseCQ [9] departs from the use of data-parallel algorithms for operations such as reductions, and instead implements pipelined versions of said algorithms using GPU atomic instructions. Kernel Weaver [33] is a compiler that automatically tries to fuse multiple relational operations together into a single kernel, in order to i) reduce data movement and ii) enable additional compiler optimizations over the fused operators. Finally, MapD [1] ports the paradigm of CPU-based query compilation [23] in the context of GPU DBMS. MapD uses the LLVM compiler infrastructure to generate the code for its kernels just in time; the kernels contain code which is specialized for the current query, and try to minimize the amount of intermediate results per query.

GPU engines on heterogeneous servers. The majority of GPU-powered DBMS adopt one point in the design spectrum and make one or more of the following simplifying assumptions: First, they rely on the input dataset being GPU-resident or copartitioned to avoid the PCIe transfer overhead for input and intermediate data [1]. Second, many support query execution on a single GPU instead of multiple ones [13]. Third, the mechanisms they use for parallelizing queries is strictly tailored for GPUs. This leaves a substantial amount of CPU-based processing capacity under utilized when used on heterogeneous servers, and misses out on potential co-processing opportunities where a query can be parallelized across CPUs and GPUs simultaneously. The few engines that do support splitting a query and executing it on both CPUs and GPUs [16] rely on wasteful full materialization.

In summary, to our knowledge, there is no abstraction today that makes it possible to parallelize compiled query plans across GPUs and CPUs in heterogeneous servers.

3. THE HETEXCHANGE FRAMEWORK

To fully utilize the capabilities of heterogeneous servers, DBMS must be able to exploit both intra-device data parallelism offered by GPUs, inter-core task-parallelism offered by CPUs, and cross-device heterogeneous parallelism across multiple CPUs and GPUs. In addition, it must exploit fast node-local memory available in CPUs and GPUs while simultaneously working around the limitation of global, cache-coherent shared memory.

HetExchange redesigns the classical Exchange operator to parallelize pipelines on multicore CPUs, multiple GPUs, and across CPUs and GPUs. In a heterogeneous parallel query execution engine, execution has to be routed between different devices. Traditionally, analytical query engines use the Exchange operator to perform such *control flow* routing between consumers and producers running on CPUs. On heterogeneous platforms, producers and consumers are not guaranteed to be of the same nature: for example, they may be CPU cores, GPUs, or a mix of CPUs and GPUs. In order to enable heterogeneous control flow transfers, HetExchange uses two control flow operators: *device crossing* and *router* operators. In addition to control flow, an Exchange operator should also deal with *data flow*, to ensure that data is transferred between producers and consumers in a pipelined fashion. HetExchange enables cross-device data flow transfers by using two operators, namely, *mem-move* and *pack*.

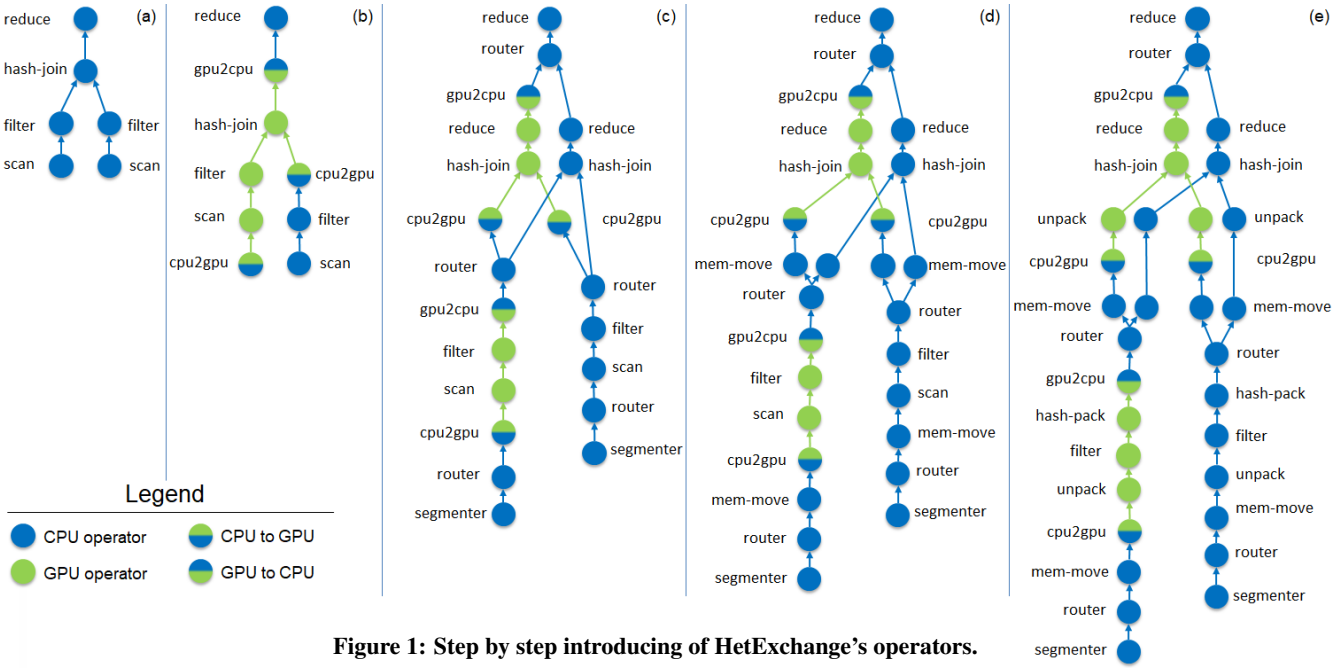

Figure 1: Step by step introducing of HetExchange's operators.

3.1 Control flow operators

HetExchange decomposes control transfers into two types: transitions between exactly one producer and one consumer of different types, and transitions between an arbitrary number of homogeneous producers and consumers. HetExchange uses two different operators, a device-crossing operator and a router, to handle each type of transfer separately. Such a separation provides a modular division of labor across the two control flow operators. Transitions between arbitrary numbers of heterogeneous units involve a combination of both operators.

Device-crossing operators enable pipelined execution across heterogeneous hardware. Except from these operators, all other operators are oblivious to hardware heterogeneity and always execute on a single device type. More specifically, HetExchange uses two device-crossing operators for CPU-GPU co-processing, called *cpu2gpu* and *gpu2cpu*. Cpu2gpu copies the CPU context to the GPU and transfers the control flow by launching a GPU kernel, while gpu2cpu transfers the GPU context to the CPU and starts a CPU task. In contrast with launching a GPU compute kernel from the CPU, GPU programming frameworks do not support launching CPU tasks in the middle of the execution, which prevents fully pipelined execution across devices. HetExchange implements this functionality by breaking the gpu2cpu operator into two parts, one that runs on each device. These parts communicate using an asynchronous queue. When a GPU kernel is ready to send a task to the CPU, the gpu2cpu operator inserts the task into the queue. On the CPU side, the second part of the operator receives it and executes it.

Router operators are used to encapsulate parallelism across multiple processors. Similar to the classical Exchange, for vertical parallelism, router operates as an asynchronous queue between a producer and a consumer. For horizontal parallelism it instantiates multiple instances of the consumers and asynchronously routes packets between the producers and the consumers. In contrast to traditional Exchange, a parallel query plan consisting with routers is essentially a directed acyclic graph. The router may have multiple parents, each of them targeting different devices. Each of the parents is instantiated multiple times to achieve the necessary degree of parallelism in each device type. The same holds for its children. The router implements various routing policies: hash-based routing for use in hash joins, round-robin/range routing for partitioning inputs to multiple consumers, and union routing for merging inputs from multiple producers.

In contrast with the classical Exchange, router only operates on the control plane. A task description refers to the target input data via a block handle. The router transfers the block handle from the producer to the consumer but not the actual data. When needed, the data flow operators handle the block creation and its transfer, as described in Section 3.2. This division of labor between the router and data-flow operators enables the router to connect producers and consumers without making assumptions about data location or accessibility.

While the router avoids data transfers by operating with block handles, in some cases, the router itself needs access to the values of each tuple. An example of such a case is the hash-based routing policy that uses the hash value of input tuples to determine the target consumer. Due to the heterogeneous nature of memory access in CPU–GPU servers, such data might not be directly accessible by the router, as would be the case for a router running on the CPU that attempts to access a GPU resident block of data. Thus, performing routing would force the router to either transfer the data to evaluate the routing policy, or operate on multiple device types to run locally with respect to the target block of data. None of the solutions is modular, as they would duplicate data movement in the router and data flow operators.

HetExchange uses an approach tailored to the heterogeneous servers to handle such cases. Instead of having the router access tuples for determining policies, HetExchange pushes the policy mechanism down to the data flow operator (described in Section 3.2) that has visibility to the data. For example, in order to use a hash-based routing policy over blocks of tuples, we require each block to have only tuples with the same hash value. This is achieved by enforcing the data flow operator that produces these blocks to maintain this invariant during the creation of each block. Each block handle provided by the data flow operator is then forwarded to the router operator with the corresponding hash value. Thus, the hash-based routing policy decides without having access to individual tuples.

Another difference between the router and classical Exchange is that the router does not perform broadcasts. Efficiently executing

a broadcast dependents on both the memory topology and the initial location of the data. For example, it may be possible to just share data between the targets or use some multi-cast capability of the interconnect. In addition, broadcasts are inherently data flow operations, as they duplicate data flow inside the plan. On the contrary, assigning the different flows to different execution streams is a control flow operation. Thus, broadcast, in the sense of data duplication, is left to the mem-move, described in Section 3.2. For this case, the mem-move operator produces as output one block handle per broadcast target and a value, the target id. Then the router routes the block based on the target id, without caring about how the data were actually broadcasted. From the router's perspective, this is similar to a hash-based policy.

Encapsulating heterogeneous parallelism: example. Combining the router with the device crossing operators creates all the necessary control flow manipulations to enable all three types of parallelism across multiple heterogeneous compute units. Device crossing operators are placed between heterogeneous producers and consumers to move execution across device types. Routers are placed at strategic points before device crossing operators to parallelize query plans. We use a reduction over the results of an equijoin of two filtered tables as a running example to illustrate how control-flow operators work. Figure 1(a) depicts a physical plan for such a query, generated for sequential CPU-only execution.

In the running example, placing three device crossing operators is enough to move the execution of the hash-join to the GPU. An example of such a plan is shown in Figure 1(b). A cpu2gpu operator is placed on the left to kick-start the execution of the left-hand scan and filtering pipeline on the GPU. This cpu2gpu operator transfers execution from the CPU to the GPU and as a result, feeds the hash-join build phase on the GPU. The scan and filter operations for the probe table are executed on the CPU in this example. As the filter selects some tuples, it forwards them to the cpu2gpu operator above it which then transfers it to the probe phase of the GPU join. Similarly, the cpu2gpu above the hash-join transfers the execution back to the CPU side for the final reduction.

Figure 1(c) extends 1(b) with router operators. The example shows five routers in order to parallelize the hash-join over all the CPUs and GPUs of the system. In the left-hand side, the segmenter will split the input file into small block-shaped partitions, that are treated as normal blocks. The block handles to these partitions will be propagated to the router, which instantiates the scan-filter-gpu2cpu consumer multiple times and routes partitions to the consumers in a load-balanced way.

Each of the GPU scans will read the partitions which are propagated to it by the router, via the cpu2gpu operator. The filter performs predicate evaluation and propagates passing tuples to its corresponding gpu2cpu operator, which in turn forwards them to the router. This router unions the results from the GPU filtering and distribute them to its consumers. This router has two parents in the plan, one of them to execute the hash-join on GPUs and the other to execute it on CPUs. Each parent is instantiated multiple times, for example, the first one as many times as the number of available GPUs, and the second as many times as the number of available CPU cores. As the results are routed between the consumers, the join ends up running in a mix of CPUs and GPUs. After the joins, a local reduction happens in each device and the output of each local reduction is send to the union router which gathers all of them into a single thread in order to produce a final global aggregation.

3.2 Data flow operators

As a heterogeneous server usually has multiple memory nodes, the query execution has to deal with issues regarding the accessibility of each operator's input. For example, GPU memory is not accessible in hardware by the CPU and in some cases, might not be accessible by other GPUs. While the control flow operators enable parallel and pipelined execution across multiple heterogeneous devices, none of them actually considers whether the input data are accessible by their consumers. HetExchange encapsulates memory-access heterogeneity using two operators, namely, mem-move and pack.

Mem-move operator. The mem-move operator is responsible for moving data between node-local memory of producers and consumers. It receives a block handle from its child, a data producer, in the query plan that contains information about the sources and targets for each data block that it must move. Using this information, the mem-move is responsible for ensuring that the data is transfered and accessible before its client, the data consumer, is executed.

Mem-move encapsulates the logic to drive the transfers over the interconnects as well as to take decisions based on the topology and the initial location of the data. In case the data are already local to the consumer, it only forwards the block handle, without doing any data transfers. In situations where a CPU producer must be connected to a GPU consumer, or vice versa, it is responsible for launching the necessary DMA transfers over the PCIe to move data from CPU host memory to GPU device memory. As the mem-move abstracts away memory heterogeneity issues, all other operators can be data-location agnostic. Thus, other operators do not have to be programmed to perform explicit data transfers or data accessibility checks. Based on the information mem-move has regarding the data flow from the query plan, it automatically prefetches data to consumer's local memory before the consumer accesses them.

Memory transfers happen asynchronously to computation. Mem-move is internally consisted of two parts, one that resides on the producer and one that resides on the consumer. When the producer's part of mem-move receives a block handle from the producer, it schedules the transfer and returns back to the producer, to allow it to generate the next block. The consumer part of mem-move waits for transfers to complete. When a transfer completes, it pushes the block to the consumer. As a result, both the consumer and the producer execute asynchronously with respect to the memory transfer. Mem-move is also responsible for multi-casting. For certain operations, like a broadcast-based hash-join, it is common that copies of the same chunk of data should be sent to multiple consumers. Multi-casting is essentially a special case of data transfer and multiple interconnects support it. Thus, in HetExchange, mem-move bears the responsibility of broadcasting and implementations can potentially exploit the capabilities of the underlying interconnects to do it efficiently.

Pack/unpack operators. Moving data is expensive and is often the bottleneck in GPU query processing. HetExchange amortizes data transfer cost by executing transfers at block granularity, instead of tuples. However, as we described in Section 2, block-at-time execution of operators on the GPU is suboptimal due to materialization overhead compared to fusing operators into few kernels using JIT compilation, and having each GPU thread perform tuple-at-a-time execution with register pipelining [9].

HetExchange uses the *pack* operators to encapsulate the difference between block-at-a-time data movement and tuple-at-a-time execution. The two basic operators of this set are *pack* and *unpack*. The pack operator groups tuples into a block and flushes it to the next operator whenever it fills up. The unpack operator takes a block of tuples as input and feeds them one tuple at a time to the next operator. HetExchange also uses the pack operator to create blocks with interesting properties. When used to pack/unpack data for a consumer that is a GPU operator, these operators ensure that the grouping of tuples enables different GPU threads to read data in a coalesced manner. When used to pack/unpack data for a hash

join, the pack operator generates blocks whose tuples have the same hash value by maintaining one block per hash value, that is flushed to the next operator whenever it's full. As all the tuples in a block have the same hash value, consumer operators, like the router, can operate over the whole block, without accessing individual tuples.

Encapsulating heterogeneous memory access example. We extend the running example shown in Figure 1(c) by placing mem-move operators in order to move the data to the point of their consumption. Figure 1(d) shows a plan that is distributing the data based on their hash values for the join. In the left-hand side of the plan, a mem-move is placed after the router responsible for distributing the input segments. As input segments are pushed from the segmenter to the router and routed to the different GPUs, the mem-move after the router will make sure that the data are accessible by the target GPU. For example, if a block is routed to a GPU but residents on another one or on the CPU, mem-move will transfer it to this GPU. If it is already on the destination node, it will propagate the block handle, without transferring data.

Figure 1(e) extends 1(d) by adding pack/unpack operators. Notably, the scan operators of Figure 1(d) are replaced by unpack operators in Figure 1(d) to highlight the fact that each unpack operator processes multiple blocks of input. In addition, as the data shuffling between the filtering and join phases is in blocks, unpack operators are placed in each device to translate between blocks and tuples. For the same reason both filters are followed by packing operators.

As the query plan uses a hash join, the packing after the filter is a hash-pack. Each time the hash-pack outputs a block, it also outputs the hash value of the block elements. In the left-hand side of the plan, the hash-pack will push the block handle and the hash value to the gpu2cpu operator, which will propagate both of them to its CPU side and then to the router, which will route the block based on the hash-value.

In this specific plan, all the consumers start with a mem-move. Thus, when a mem-move receives a block handle, it transfer the block data to the target device if necessary. Then, mem-move forwards the handle to the cpu2gpu operator. Cpu2gpu will launch a kernel to consume this block, which will start by distributing and scanning the block to the different GPU thread using the unpack.

3.3 Integration with the query optimizer

Query execution on heterogeneous hardware has four fundamental traits: target device, degree of parallelism, data locality and data packing (whether chunks of data can be sequentially accessed). Each of the four operator of the HetExchange framework changes one of these traits on its output, without modifying its input. The device crossing operators change the target device trait and the router changes the degree of parallelism while the mem-move operators change the data locality and the pack/unpack operators change the packing. Existing work [3, 11] has already focused on supporting physical properties/traits and converters: operators that do not modify their data, but only guarantee specific properties for their output, such as the sort operator which does not modify its input but guarantees an order for its output. The proposed operators are essentially new converters and thus compatible with such systems. From the query optimizer perspective, the relational operator will then require their input to have two traits: be local and unpacked.

The same separation of concerns assists in cost modeling the new operators. The cost of the device crossing operators is the cost of spawning a task from the source device to the target one. The router has the cost of routing the packets, without transferring any data. Data transfers over the interconnects are modeled as the cost of the mem-move operators and lastly the pack/upack operators measure the cost of scanning and materializing intermediate results in blocks. Due to the heterogeneous hardware, the cost of the relational operators depends on the target device, similarly to how the cost of relational operators depend on whether their input is sorted.

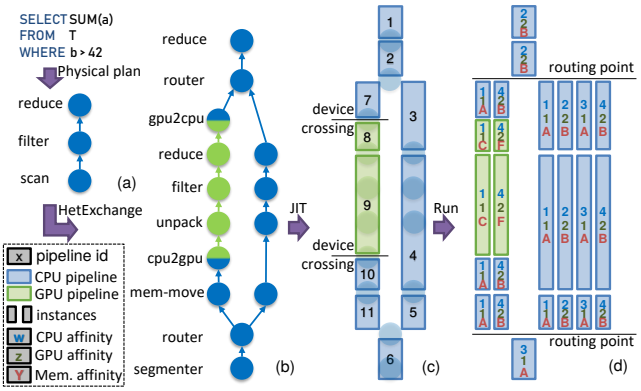

Figure 2: Pipelines and affinities of a hybrid plan.

The router-related transformation rules resemble the ones of the classical Exchange; the rules differ only in terms of the use of broadcast/hash-based Exchange, which HetExchange replaces with a broadcast mem-move/hash-pack, respectively, before the router. As for operator placement in the query plan, device crossing operators can be placed anywhere in the plan and pushed up or below any other operator, with the exception of mem-moves and routers. On the other hand, while pack/unpack operators can be placed anywhere, they can only be pushed up or below other HetExchange operators. In addition, pack/unpack operators can be used as staging points, similarly to [22], in order to improve vectorization in the CPU side and re-convergence in the GPU side. Mem-moves are inserted to fix data locality before the flow reaches relational operators, but usually they run on the CPU side. Assuming an optimizer that can represent the different traits, such as the one of Apache Calcite, these rules are straightforward to integrate.

4. HETEROGENEOUS JIT COMPILATION

While HetExchange is a general abstraction, in this section we present its interaction with code generation in order to enable a JIT DMBS to parallelize query execution on heterogeneous hardware.

This section discusses the lifetime of a query in a HetExchange augmented JIT DBMS and uses an aggregation over a filtered table as a running example. Figure 2 depicts the different stages in the lifetime of the query. When the query is submitted, it is first converted into a physical plan, agnostic to the heterogeneity and parallelism of the server, shown in 2(a). The physical plan is then augmented with the HetExchange operators, described in the previous section, in order to produce a heterogeneity-aware plan. The resulting plan is shown in Figure 2(b) and it parallelizes the query over the mix of CPUs and GPUs available on the system. Then, through JIT compilation the DBMS produces machine code specialized to the server's devices as described in 4.1. When invoked, the generated code controls the number of instances of its different parts and cooperates with the memory subsystem in order to efficiently utilize the server, described in 4.2 and 4.3.

4.1 Generating heterogeneous pipelines

During code generation the query plan is split into pipelines, and specialized code is generated for each pipeline. Operators that force materialization of intermediate results are typically called *pipeline breakers* [23], and produce code that i) materializes results emitted by the pipeline before the breaker operator, and ii) triggers result iteration in the pipeline after the breaking point. As HetExchange operators are handling execution over multiple devices and

Device Provider Methods		
allocStateVar	get/releaseBuffer	#threadsInWorker
freeStateVar	malloc/free	threadIdInWorker
storeStateVar	convertToMachineCode	loadMachineCode
loadStateVar	workerScopedAtomic<T, Op>	

Table 1: Methods overwritten for each targeted device by the device crossing operators

memories, they are inherently pipeline breakers: they have to materialize the results out of the registers into memory. Still, they emit output in batches, without having to first process the entire input.

The code generation phase outputs a set of pipelines, where each pipeline is the result of fusing the operators between pipeline breakers into tight segments of code. Pipelines corresponding to the leaves of the query plan trigger the entire generated code; every other pipeline is invoked as a result of invoking these ones.

Traditionally, in a JIT DBMS, operators are code generation modules that expose two functions [23, 17]: *produce()* and *consume()*. *Produce()* is called recursively by every operator in top-down fashion (i.e., starting from the root of the query plan): every operator asks its children to *produce* their result tuples. *Consume()* is called recursively by every operator in bottom-up fashion: every operator asks its parent to consume the tuples just pushed to it, essentially asking the parent to generate its physical implementation.

In the running example, when the router at the bottom of the plan is about to generate code, it will call the produce() method of the segmenter, so that the latter generates its code first. The segmenter is a leaf operator, so it will proceed with code generation without further produce() calls. Instead, the segmenter will generate code similar to lines 1–3 of Listing 1: The segmenter's generated code comprises a nesting of two loops, which gather the list of memory segments of relation T, and break them into blocks. Then, the segmenter will call the router's consume() method, triggering the router to produce its physical implementation. The router will then produce its implementation (lines 4–5), evaluating the policy on each block, and based on the result sending the block handle to a specific consumer that is either an instance of pipeline 5 or 11.

JIT on multiple devices: The missing pieces. Directly mapping traditional JIT techniques to the case of heterogeneous servers would require having multiple implementations of the same high-level operators, with each implementation targeting a different device. Such a design is inconvenient and inflexible, causing increased programming and maintenance effort. For example, a relational select operator would require a different implementation and code generation procedure per device, thus hindering the extensibility of such an architecture. In addition, in order to achieve inter-device task-parallelism, the JIT infrastructure has to be able to handle transitions between different device type targets; otherwise, the generated code will target only one device type.

JIT on multiple devices with HetExchange. HetExchange simplifies multi-device code generation in three steps: First, it decomposes the query plan into multiple parts, each of which is specific to a device type. Second, the aforementioned device crossing operators of HetExchange also encapsulate the transitions between compilation targets. Finally, HetExchange redesigns the *produce()* and *consume()* methods of each operator to enable them to generate code that is device-specific, yet not specializing their implementation to a device. To achieve this generality, HetExchange parameterizes each method with a device-specific *provider*.

Device providers. Even if a JIT DBMS generates code for a single device, it should ideally rely on a collection of utility functions as building blocks for its implementation. These utility functions should handle operations such as the following: i) Locating a pipeline's state, such as pointers to data structures ii) Acquiring/releasing device memory iii) Acquiring/releasing locks, and performing atomic operations iv) Retrieving device-specific characteristics, such as the grid and block size used by a GPU kernel.

```
1  def pipeline6()
2    for each segment in file
3      for each block in segment
4        c ← evaluate policy on block
5        send handle of block to consumer c
6
7  def pipeline11()
8    for each received block handle b
9      if b not on destination
10       d ← get block handle on destination
11       schedule DMA copy from b to d
12       send d to inst
13     else
14       send b to inst
15
16 def pipeline10()
17   for each received block handle b
18     wait DMA transfer for b to finish
19     schedule pipeline9(b) for GPU execution
20
21 def pipeline9(data_block[N], state)
22   local_acc ← 0
23   for i = threadIdInWorker to N − 1 with step
         #threadsInWorker
24     t ← data_block[i]
25     if t.a > 42
26       local_acc ← local_acc + t.b
27   nh_acc ← neighborhood_reduce(local_acc)
28   if thread neighborhood leader
29     atomic_add(state.acc, nh_acc)
```

Listing 1: Pseudo-code for pipelines 6 and 9-11.

HetExchange groups the collection of all the utility functions into a device-independent interface, and offers a collection of *device providers* implementing said interface; a CPU- and a GPU-specific handler at the moment. Device crossing operators are the ones specifying which device handler every pipeline should use; each pipeline's operators then use the handler provided to them when appropriate. Thus, if a pipeline targets, for example, a GPU device, the methods of the pipeline's operators will make calls to a *GPU provider* in order to generate GPU-specific stubs. The same pipeline could generate code for a CPU with no changes other than being instantiated with a different handler as input. The overall implementation of the produce() and consume() methods per operator will thus remain agnostic to the device properties.

Aside from their other responsibilities, the device providers also guide the final steps of the compilation in order to optimize the generated code and produce machine code for the target device. Upon completing code generation of a pipeline, it is optimized, compiled down to machine code and loaded into the running instance of the DBMS. The device provider of each pipeline is responsible for specifying how each of these steps is achieved.

JIT code for heterogeneous servers example. As already described in the running example of Figure 2(c), the segment and the producer part of the bottom router will be fused into pipeline 6, that sends block handles to pipelines 5 and 11. Both of these pipelines wait for handles from the router, as part of the code generated by the consumer part of the router. Then, mem-move will generate code that checks for each received handle if the block is on the target memory node. If it isn't, the generated code requests a new block on that node and spawns an asynchronous DMA transfer to copy the data to it. In any case, mem-move propagates to the next pipeline a block handle that is on the local-to-the-consumer memory node together with information about which transfer the consumer should wait for, if any. In the beginning of pipelines 10 and 4, the two mem-moves inject code to receive these handles and wait for the transfer to complete (lines 16–18 of listing 1). Then,

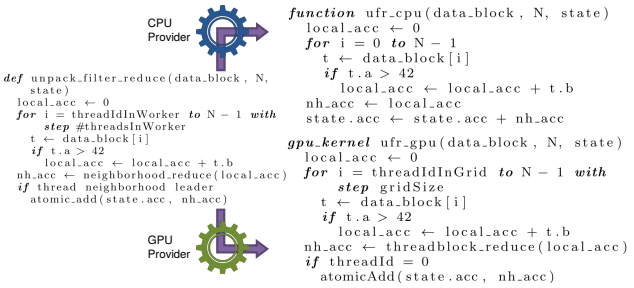

Figure 3: Providers specialize code to the target device type.

pipeline 4 will unpack the block, check the filter and update the accumulator, based on the code generated by the unpack, filter and consumer part of reduce respectively. On the other hand, pipeline 10 will schedule a GPU kernel of pipeline 9 with the received block as argument, due to the code generated by the producer part of the cpu2gpu operator.

Listing 1 shows, in pseudocode, a simplified version of the generated code for pipeline 9 of the running example. The four participating operators are fused into a simple GPU kernel that scans each block, evaluates the filtering predicate and increments the accumulator accordingly. The consumer part of the cpu2gpu operator specifies the arguments of the pipeline and the unpack generates the scanning. For each tuple, the code generated by the filter and the reduce in lines 25–29 is executed.

Pipeline 8 will read the final result of the aggregation and insert it in the queue of the gpu2cpu operator. On the consumer side, the gpu2cpu operator generates code to wait for input in the queue and when values are written, it reads them and propagates them to the router, which will send them to its single consumer, the single instance of pipeline 2. Similarly pipeline 3 reads the result of the CPU reduction and sends it to the same instance of pipeline 2 via the router's queues. Pipeline 2 waits for the partial aggregations to arrive via the router and accumulates them. Pipeline 1 will read the final aggregation that is the results of query.

In the running example, pipeline 9 is associated with the GPU provider, as it targets GPU execution. The provider will translate threadIdInWorker into the id of the thread inside the kernel, while *#threadsInWorker* will be translated to the number of GPU-threads used by the kernel. When memory is allocated for the *global_acc* accumulator in the state, the provider will generate a call to the GPU memory allocator in order to allocate the state in GPU memory. In addition, the neighborhood considered by *neighborhood_reduce* will be a GPU thread-block and the worker-scoped atomic add will be translated into the corresponding GPU atomic instruction. Lastly, the provider will optimize the pipeline after its generation, then compile it down to machine code for the GPU and load it into the GPUs. Pipelines 9 and 8 target GPU execution and thus are associated with the GPU provider. All other pipelines are associated with the CPU provider.

Figure 3 shows an example of how the same pipeline results into different code depending on the provider it uses. The pipeline depicted on the left-hand side of the figure is provider-agnostic, and generic enough to be specialized for a CPU or a GPU. The code loops through thread workers with increments of a given step, evaluates a filtering condition, and increases the value of a thread-local variable when the condition is successful. Once the loop has completed, the operator accumulates the thread-local variables into per-warp variables, and then the leader of each warp updates a worker-scoped accumulator atomically. If HetExchange was not using device-specific providers, the code they will produce when specializing the left-hand pipeline into pipeline 4 and 9 will be similar, and actually suboptimal for CPU execution, because it would be overly complex. Instead, through the use of CPU and GPU providers, HetExchange specializes code to the target device, while keeping the operator "blueprints" the same for both devices: For example, the *threadIdInWoker* will be set to 0 for the CPU provider, while it will be set to the GPU grid-wide thread id for the GPU provider. Similarly, the *#threadsInWorker* will be set to 1 for the CPU provider and to *gridSize* for the other one. More importantly, as there is a single thread in the CPU case, the worker-scoped atomic and the neighborhood-local reduction will be optimized out.

4.2 Controlling parallelism and affinity

In a HetExchange augmented DMBS, the router controls the horizontal degree of parallelism for the query plan operators above it. At code generation time, depending on its policy (e.g. hash-based, round-robin, etc.) and the intended degree of parallelism, the router is responsible i) for producing multiple pipelines on its consumer side, and ii) for triggering code generation for these pipelines. An additional source of complexity is that while the classical Exchange has one parent and one child that are instantiated multiple times, the router has multiple parents and children in order to parallelize the rest of the plan to a mix of compute units.

Given that the pipeline instances to be generated are almost identical, it would be inefficient to trigger code generation from scratch for every one of them. Thus, the router generates a parameterizable version of the pipeline in question *per device* (instead of per thread), and then initializes multiple instances from this "pipeline template" (i.e., performs state creation for each one).

As only the router controls parallelism, it is also responsible for pinning pipelines to specific devices, based on pluggable policies. When a router instantiates its consumers, it locks them to specific devices. In order for policies to be able to control pipelines not attached to a router (e.g. pipelines 9 & 4 of the running example), HetExchange forces pipelines to inherit both degree of parallelism and the affinity of their instantiator. Assigning both a CPU and GPU affinity to all pipelines, but using only the appropriate one, allows routers to control the affinity of pipelines even after multiple device crossings (e.g. the bottom router controls the affinity of pipeline 7; the information is not lost by the device crossings).

In its current form, the router specifies operator affinity, degree of parallelism and routing policy statically at query time. Future work involves making such decisions dynamically and integrating existing work in this direction such as dynamic schedulers [32] and opportunistic task stealing between different pipelines [21].

Parallelism and affinity example. As in the running example of Figure 2(d), pipeline 6 is a leaf pipeline, it runs single-threaded. The bottom router injects code in pipeline 6 to instantiate pipelines 11 and 5, two and four times respectively. In addition, the router will affinitize the first instance of pipeline 11 to CPU core 1 and GPU 1, and the second instance to core 4 and GPU 2. Each instance of pipeline 11 will create an instance of pipeline 10 and the latter will copy its instantiator's affinity. Similarly, pipelines 7-11 will have two instances with the corresponding instances affinitized to the same compute units. The GPU affinities will be considered only for pipelines 8 & 9, while all other pipelines use the CPU ones.

4.3 Memory management and data transfers

During query execution, memory is used either to store state of operators, like the hash table of a hash-join, or to stage blocks of intermediate results before transferring them between devices. HetExchange distinguishes between the two and has a different manager for each of them. State memory is served by memory managers, while staging memory is served by block managers. Both memory and block managers are organized as a set of indepen-

2 × Intel Xeon CPU E5-2650L v3 @ 1.80GHz, 12 cores per CPU
256GB RAM (128GB per socket), 66% of memory slots populated
64KB L1 (32KB L1d, 32KB L1i) per core, 256KB L2 per core; 30MB L3 shared
2 x NVIDIA GeForce GTX 1080 with 8GB memory per GPU

Table 2: 2 Socket, 2 GPU machine used for the experiments.

dent, local components – one per memory node. Requests by the pipelines are always served by their closest (appropriate) manager.

While memory managers only manage local memory, block managers frequently handle data operations that involve remote devices. In addition, block managers need to be thread-safe, yet existing synchronization primitives are very expensive due to the absence of global, cache-coherent shared memory. HetExchange tackles these challenges in the following ways: Firstly, at system initialization time, the block managers pre-allocate memory (block) arenas, to avoid memory allocation costs at query execution time. Secondly, to circumvent the absence of coherence, HetExchange allows only local devices to acquire blocks from a block manager and opts for device-local synchronization primitives. To serve requests for remote blocks, the block managers launch small tasks to the corresponding node to acquire blocks. As this can become expensive, HetExchange accelerates the common cases by i) having each local block manager maintain a cache of acquired blocks per remote manager, and ii) batching requests for block acquisition and release from remote nodes.

5. SYSTEM

We integrate HetExchange and its system architecture to Proteus [17], an analytical query engine that utilizes LLVM-based code generation. Proteus originally generated CPU-specific and single-threaded code. Therefore, we extended Proteus's infrastructure to allow GPU-specific code generation, by introducing code generation components for single-GPU operators. Enabling Proteus to operate over multiple CPUs and GPUs requires i) extending its code generation infrastructure to produce code for parallel execution, and ii) coupling Proteus with HetExchange non-intrusively.

LLVM is capable of compiling code for multiple architectures by using a different back-end for each target, like the x86_64 back-end that is used by Proteus for code generation targeting Intel CPUs. In addition, LLVM has back-ends for both NVIDIA and AMD GPUs. For the evaluation of HetExchange we use the NVPTX back-end to generate code for NVIDIA GPUs. While our methods are applicable to AMD GPUs, we leave the implementation as future work.

During code generation, the operators generate code via the device providers. In Proteus, the providers use LLVM's code generation interface for the low-level code generation, such as load and store operations, while for the high-level functionality, like state manipulation and memory allocations, they are emitting the relevant code. The generated code is optimized using LLVM. The CPU provider also uses LLVM to compile the IR down to machine code and loads it in the running instance, while the GPU provider uses LLVM to compile the IR down to PTX[24], an assembly language for NVIDIA GPUs, and the CUDA driver API to compile PTX to machine code and load it to the GPUs.

Similarly to Figure 2, upon receiving a query, the extended Proteus parses and optimizes it in order to produce a single-threaded CPU-only physical plan, like the one in Figure 1a. This plan is then extended with the HetExchange operators to a heterogeneity-aware plan like the one in Figure 1e. The heterogeneity-aware plan describes which devices will be used in each part of the generated code. Then, based on the heterogeneity-aware plan, Proteus generates code for the query and start executing it. In our implementation, part of the query optimization is handled by Apache Calcite[3]. We opted for this three step query optimization process (logical → physical → heterogeneity-aware plan) as a proof of concept, but integrating the two last steps into a single one is also possible. Selecting between this two options should yield a trade-off between plan optimality and query optimization times. While producing heterogeneity-aware plans is a topic worth as much research as enforcing them, we leave it as future work and for this evaluation we heuristically add the HetExchange operators. For this work, we opted for the three step process, as between these two options, it creates the smallest overhead to the query optimizer.

6. EXPERIMENTAL EVALUATION

Experimental Setup. We compare Proteus against state-of-the art commercial analytical engines DBMS C and DBMS G for CPU and for GPU execution. DBMS C is a columnar database that uses SIMD vector-at-a-time execution, similar to MonetDB/X100 [4], and supports multi-CPU execution. DBMS G uses JIT code generation, operates over columnar data and supports multi-GPU execution. We use various configurations of Proteus (i.e., CPU-only, GPU-only, and hybrid execution) to showcase its versatility and its ability to execute queries efficiently regardless of where data is originally located – i.e., the CPU or the GPU memory. We warm up each system by executing multiple queries before the measurements. The experiments run on the machine described in Table 2. Each socket has one GPU attached via a dedicated PCIe 3.0 x16 connecting it to the local socket. We measure a maximum bandwidth of ∼12GBps on each interconnect, on an idle server.

Similar to prior work on GPU DBMS [5, 34], we use the Star Schema Benchmark [25] to compare three configurations of Proteus against DBMS G and DBMS C. Proteus GPU and DBMS G use the two GPUs of the server, Proteus CPU and DBMS C use the two CPU sockets and Proteus Hybrid uses both the GPUs and the CPU sockets. For Proteus Hybrid we select plans that parallelize all the relational operators across all the CPUs and GPUs available. While is possible to pin parts of the plan to specific processors, we leave optimizer-driven plan generation with different parts of a plan running on different processor sets as future work.

We use two scale factors, SF100 and SF1000, in order to examine the behavior of the two GPU-powered DBMS, Proteus GPU and DBMS G, when the input data fits in the aggregate GPU memory (SF100) and when it does not (SF1000).

6.1 GPU-fitting data (SF100)

Methodology. Proteus GPU and DBMS G fit the table columns that the SSB queries touch in the aggregate device memory of the two GPUs (16GB). DBMS C and Proteus CPU configurations operate over columnar data that reside in the CPU memory.

The Proteus GPU configuration randomly partitions each table between the two GPUs. We profiled DBMS G and noticed an absence of cross-GPU PCIe traffic during query execution, therefore DBMS G either performs co-partitioning of the fact and the dimension tables, or broadcasts (dimension) tables to both GPUs a priori. For all queries, the optimizer of Proteus opts for broadcast-hash-join-based plans; the HetExchange operator broadcasts the necessary columns of dimension tables involved in joins to both GPUs. DBMS G opts for a star-join-specific join implementation: It conceptually treats each dimension table as a dense array $dimtable[]$, where the value $dimtable[key_i]$ corresponds to the tuple whose key column value is key_i. DBMS G performs the (star) join by iterating over the fact table's values, and finding the corresponding values from the dimension tables/arrays via array index lookup.

Figure 4 depicts results for SF100; the HetExchange operators enable Proteus to seamlessly parallelize its execution across computational units. Q1.1 - Q1.3 are the simplest SSB queries; they perform a single join of the lineorder fact table with the dates table.

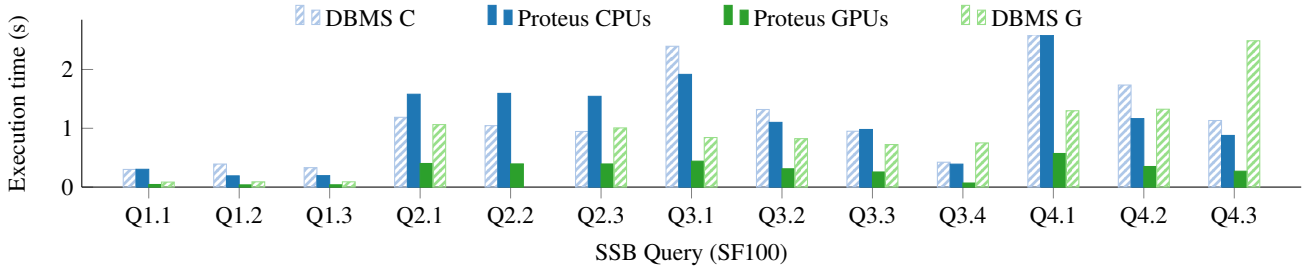

Figure 4: SSB with GPU-fitting working sets. Data in GPU memory GPU systems.

Proteus GPU and DBMS G outperform the CPU-based systems, because the GPU devices offer high memory bandwidth (320GB/s) and number of hardware threads. Proteus GPU utilizes the resources of the GPU devices more efficiently and thus outperforms DBMS G. Specifically, every execution unit that DBMS G triggers on the GPU devices allocates double the number of GPU registers than Proteus GPU. As a result, DBMS G launches fewer simultaneous execution units, thereby underutilizing the large number of available GPU hardware threads.

Q2.1 - Q2.3 increase the number of joins between the fact table and dimension tables to three; the effect of hardware underutilization becomes more visible for DBMS G, thus its difference from Proteus GPU increases, and its performance resembles that of DBMS C. DBMS G fails to execute Q2.2's string inequalities.

Q3.1 - Q3.4 also have three joins, with each consecutive query being more selective than the previous; Proteus GPU is consistently faster. For Q3.1 and Q3.2, Proteus CPU is faster than DBMS C because the operators of DBMS C have to either materialize a result vector or a bitmap vector, whereas Proteus CPU attempts to operate as much as possible over CPU-register-based values to avoid materialization costs. Q3.3 and Q3.4 are more selective, therefore the gap between Proteus CPU and DBMS C becomes minimal. In addition, although the star join implementation of DBMS G turns joins into inexpensive array lookups, DBMS G also opts to apply filtering predicates *after* the completion of the star join, so that the dimension tables resemble sorted, dense arrays at join time, and the star join turns into a sequence of array index lookups. Thus, DBMS G's benefit from selective filtering predicates is minimal.

Q4.1 - Q4.3 increase the number of joins to four, with each consecutive query being more selective, and are the most challenging part of SSB. All systems except DBMS G benefit from queries being more selective. The Proteus configurations outperform their CPU/GPU counterparts due to the minimal generated code that comprises every query pipeline that Proteus executes and the better utilization of GPU hardware resources.

Summary. HetExchange enables Proteus to parallelize queries across multiple CPUs and GPUs and operate over different initial data placements, in the same infrastructure, without loss of generality or performance. Proteus is comparable or outperforms state-of-the-art DBMS that target CPUs or GPUs. When the working set fits in the aggregate GPU memory, Proteus achieves up to 2x and 10.8x versus CPU- and GPU-based alternatives, respectively.

6.2 Non-GPU-fitting data (SF1000)

Methodology. We use SF=1000 for SSB which generates ∼600GB of data. For all the queries the working set exceeds the aggregate device memory of the two GPUs. Thus, both Proteus GPU and DBMS G transfer from CPU to GPU memory the working set, during query execution. As a result, their throughput is upper bound at the PCIe bandwidth (∼24GBps), represented by the dotted line in Figure 5. Proteus Hybrid load balances the work between the GPUs and CPUs and thus transfers only part of the dataset to the GPUs. While Proteus supports datasets that are partially preloaded in GPU memory, we disable this functionality for this experiment to simulate worst-case transfer times. Figure 5 plots the results.

Proteus GPU achieves ∼21GBps CPU-to-GPU bandwidth for all the queries except Q3.1, efficiently utilizing the interconnects. In Q3.1, the increased selectivity of the first joins of the query increases the number of probes in the next joins and the random accesses start to become a bottleneck, reducing the performance to 16GBps. In addition, the HetExchange successfully pipelines transfers and execution and, combined with the efficient generated code, manages to completely overlap them.

In contrast, DBMS G does not reach the interconnect's throughput. DBMS G is not optimized for non-GPU resident datasets and places the dataset into pageable memory, which limits the achievable transfer bandwidth to less than half of the available for Q1.1 - Q1.3. As a solution, DBMS G proposes to use enough GPUs to fit the working set in GPU memory. For SF1000 and GPUs like the ones used in the experiment, this translates to 9-15 more GPUs.

For Q2.1-Q4.2, the DBMS G underutilizes the GPUs for the same reasons as in the previous section. For Q2.2, DBMS G reverts to CPU-only execution and takes more than 1 hour to complete, while for Q4.3 it fails to perform a cardinality estimation that is required to execute the query, due to insufficient GPU memory.

The two CPU-only systems achieve similar performance, and their trends follow the ones of SF100. In contrast with the previous experiment, the GPU systems are bounded by the data transfers. Thus the CPU systems outperform the GPU ones, whenever they can achieve higher throughputs than the interconnects. For SSB, both Proteus CPU and DBMS C only manage to overcome the 24GBps mark for Q1.1-Q1.3 and Q3.4, thus in most queries Proteus GPU prevails. The dimensional table joined in the single join of queries Q1.1-Q1.3 is small enough to fit in the caches of the CPU and thus the CPU systems achieve a throughput of 38-72GBps, or 1.5x-3x the throughput the two GPUs can access the CPU-resident datasets. Similarly, the very high selectivity of Q3.4 allows both DBMS C and Proteus CPU to exceed the 24GBps landmark and thus run faster than their GPU counterparts.

HetExchange allows Proteus Hybrid to parallelize its execution across all the CPUs and GPUs of the system and benefit in each case from using the most appropriate compute units. In queries that Proteus CPU and Proteus GPU exhibit a significant performance difference, Proteus Hybrid's execution times are close to the fastest one, as most of the load will be directed to the fastest compute units. The highest speed-ups for Proteus Hybrid are achieved when Proteus CPU and Proteus GPU exhibit similar performance, as in Q4.3. In these cases HetExchange balances the load evenly between the CPUs and GPUs of the server. In contrast with Proteus GPU, Proteus Hybrid is not bounded by the transfer time, as part of the load is served by CPUs.

In addition, we measure the throughput of the three configurations of Proteus as the size of the working set over the execution time. On average, Proteus Hybrid throughput is 88.5% of the sum of the throughputs of Proteus CPU and Proteus GPU, showing that HetExchange successfully manages to distribute and balance work

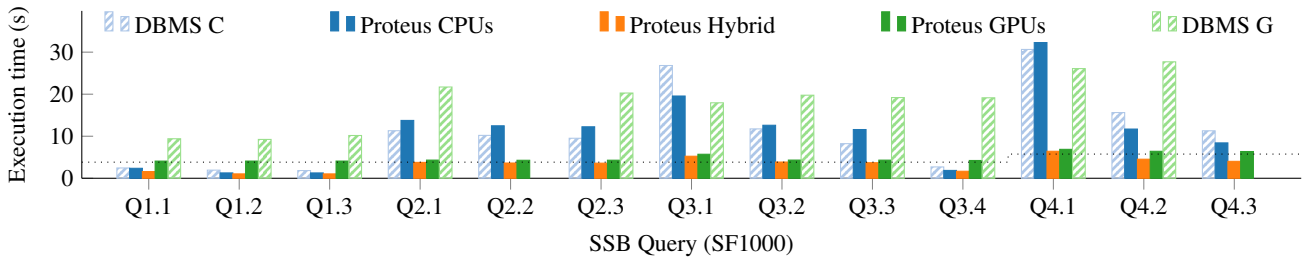

Figure 5: SSB with non-GPU-fitting working sets that are pre-loaded in CPU memory for all systems.

between the heterogeneous compute units.

Summary. HetExchange allows Proteus to use the available CPUs and GPUs efficiently, as well as combine them. In addition, HetExchange allows efficient use of the interconnects while the JIT compilation allows efficient code to be generated for each device. Even for working sets that do not fit in GPU memory, when Proteus is restricted to specific types of devices its performance is comparable or better than state-of-the-art DBMS specialized for these devices, while, when running unrestricted, Proteus Hybrid outperforms both DBMS in all the queries of SSB SF1000. Specifically, by using all the available devices Proteus Hybrid achieves 1.5-5.1x and 3.4-11.4x speed-up against the CPU-based and GPU-based homogeneous DBMS respectively, and up to 5.6x and 3.9x against its own CPU- and GPU-restricted configurations.

6.3 Scalability

Methodology. For SF=1000, we measure the total execution time for each SSB query group and different configurations of Proteus. Figure 6 plots the speed ups compared to the single threaded execution for the same query group. For all the measurements, we interleave the CPU cores between the two sockets and on the x-axis we report the degree of parallelism on the main part of the query.

For the CPU-only configurations we observe almost linear scalability up to approximately 20 CPU threads and a very limited interference when reaching the number of physical cores, due to lightweight threads like the segmenter at the bottom of the plan.

Group 1 has the best scalability for the CPU-only configurations with an average coefficiency of 87.5% per CPU core, due to its simplicity and the small cache-friendly size of its join's build side. The worst scalability is achieved by query group 2, with a coefficiency of 65% per CPU core, due to the high selectivity of its joins. Groups 3 and 4 achieve a coefficiency of 74% and 77%, respectively.

In all the cases, enabling Proteus to use the GPUs, improves performance. Query group 1 exhibits the smallest relative improvements, as the GPUs provide a relatively limited support on its effective utilization of the CPU resources and its high throughput. Two GPUs provide a speed up similar to 8-10 CPU cores. Query groups 2-4, have a higher relative performance improvement when adding the two GPUs. It provides the equivalent of adding 3.5 to 5 extra CPU sockets. The joins in groups 2-4 achieve a lower CPU throughput than the interconnect and therefore these queries have a greater benefit than the queries in group 1 from additional GPUs.

Summary. HetExchange improves performance across all query groups almost linearly as the number of CPU cores assisting the GPUs are increased, up to approximately 16 cores. For query groups 2-4, the benefit of adding more than 16 threads is offset by the interference they cause to threads that handle memory transfers and kernel launches. Using CPU cores is more efficient in query group 1, and therefore this group's performance continues to scale.

6.4 Microbenchmarking

Methodology. In the rest of this section, we micro-benchmark Proteus to evaluate the efficiency of HetExchange. Our evaluations

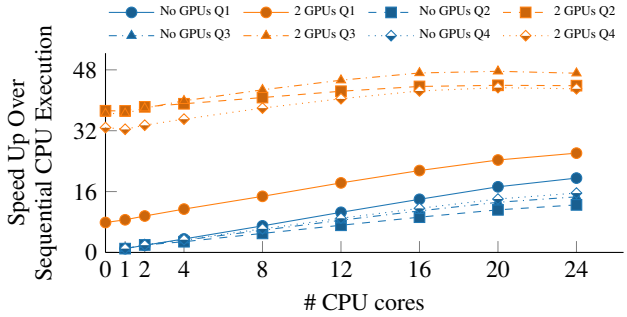

Figure 6: Scalability of Proteus.

uses two queries: i) a sum over a column and ii) a count of the results of a non-partitioned 1:N join. The first query is bandwidth intensive and thus CPU-friendly, as the GPU is behind the much-slower-than-memory-bus PCIe. The second query is GPU-friendly, as the random accesses impact the CPU side more than the GPU side. We use single-column inputs for the queries to stress out HetExchange overheads. For all the cases, the dataset is loaded and evenly distributed to the sockets. Non-HetExchange GPU Proteus overlaps transfers and computations using UVA, as in [34].

Scale-up In the first microbenchmark we measure HetExchange's execution time for the two queries and plot the results for different combinations of CPU and GPU degrees of parallelism in Figure 7. For the sum query and the probing side of the join query we use a single column of 23GB, while the build side of the join uses a 7.7MB column. We repeat the experiment and measure the execution time of Proteus without the HetExchange operators, using only its JIT infrastructure and executing on a CPU and a GPU. The results are plotted presented using dashed lines that extend to all the degrees of parallelism to emphasize the functionality provided by the proposed operators: without them, Proteus does not scale up.

When executing on a single CPU or on a single GPU, Proteus exhibits a very small overhead for using the operators. For the sum query, the HetExchange augmented Proteus scales almost linearly up until approximately 16 cores. At more than 16 cores, it operates with an input throughput of 89.7GBps which is very close to the maximum theoretical memory bandwidth we obtain from the machine (90.6GBps), given that only 66% of the memory slots are occupied. When adding the two GPUs we observe an increase in throughput of approximately 19GBps, which slowly diminishes as we increase the number of CPU cores because we exhaust the available input memory bandwidth, yielding the same peak performance when the Proteus is trying to use the whole server When using only one GPU, the peak throughput is lower, as the routing policy schedules some blocks residing on the remote-to-GPU socket to the GPU and thus causes interference to the intermediate socket.

In the join query where the performance is bottlenecked by random accesses, HetExchange scales better as CPU/GPU resources are increased. Adding a single CPU core to the GPU-only configuration causes a performance drop as GPUs have to wait for the CPU

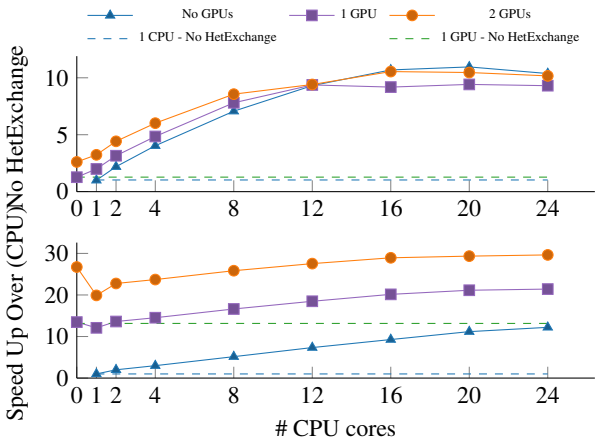

Figure 7: Scalability of Proteus on sum (top) and join (bottom)

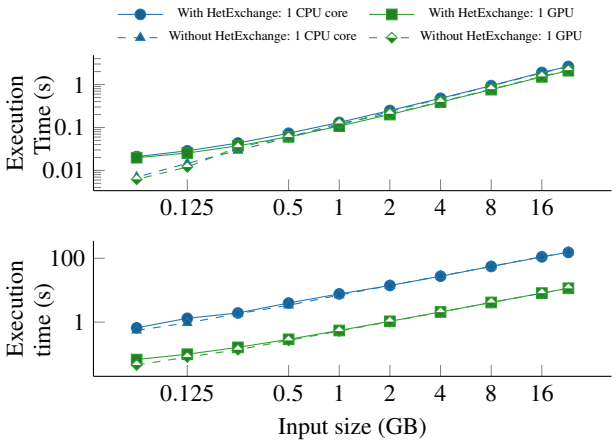

Figure 8: HetExchange for DOP=1. Top: sum, Bottom: join

hash-join's building phase, which is not replenished by the added performance of a single core. Adding cores eventually pays back, especially in the single-GPU Hybrid mode.

Size-up We repeat the experiment for the same queries, but this time we zoom-in on the overheads of HetExchange in sequential execution, to stress even more the framework. We compare Proteus without HetExchange against itself with HetExchange enabled, and vary the input size. For the HetExchange-enabled configuration, we force the optimizer to add all the HetExchange operators, despite that normally it would avoid routers for sequential execution. We restrict the degree of parallelism for the routers to 1, to match the sequential execution mode of the bare Proteus. For the join query we keep the build table size fixed to 7.7MB as in the previous microbenchmark. We plot the results in Figure 8.

In both queries, the performance is almost identical (at most 10% relative difference) for input sizes more than 512MB, as the overheads of the operators are amortized due to their block-at-a-time nature. For smaller inputs sizes 512MB and below, the difference is increased by up to 50% in the case of the summation query on the GPU and an input size of only 64MB. In these small input sizes, the high throughput of the generated code makes our current implementation of router's initialization and thread pinning (that take ∼10ms) to become a significant overhead. Allowing the optimizer to remove the router as it would do normally for such a small input, yields identical performance between the two Proteus flavors.

Summary. HetExchange allows Proteus to scale up and use the available hardware resources and our microbenchmarks show that it adds only a minimal overhead, visible only for small input sizes.

7. CONCLUSION & PERSPECTIVES

Designing HetExchange and incorporating it into a real system required considering a number of seemingly orthogonal challenges, related to i) encapsulation of parallelism, ii) encapsulation of hardware heterogeneity, and iii) choice of execution model for analytical queries; tackling these challenges led us to a number of observations that can be useful as guidelines to database system architects.

Separation of concerns. The design space for a system that can execute queries over both CPUs and GPUs is significantly wide. Picking and changing the degree (and type) of parallelism, transferring data between processors, handling arbitrary data placement across the memory of each processor, are few of the concerns to be resolved. HetExchange deals with this design space explosion by enforcing a clear separation of concerns: Explicit operators deal with orthogonal issues such as cross-device transfers, parallelism encapsulation, and memory affinity. Such compartmentalization allows the overall system to be generic, but also extensible; extending HetExchange to support another type of heterogeneous processors in the future would be non-trivial had we chosen a monolithic design. In contrast, the current design only requires an additional device provider and two device crossing operators.

Vectorization vs. compilation. Despite being coupled with a JIT compiled architecture in this work, the ability of HetExchange to enable execution over heterogeneous processors are applicable to any type of query execution engine, be it interpreted or compiled. Still, implementing a real-world system required considering a type of execution engine to pick. Given the performance benefits that they bring in analytical query processing, our two main considerations were vectorized [4] and pipelined, compiled engines [18]. If HetExchange targetted CPU processors exclusively, vectorized execution would have been a great fit as well, as there are families of operations for which it can even outperform compiled execution [30, 19, 22]. In addition, implementing a vectorized engine is more straightforward compared to implementing a JIT compiled one. However, vector-at-a-time execution can be wasteful in the context of GPU processing; the materialization overhead it entails becomes more pronounced when (cache) memory is scarce. In addition, relying on code generation infrastructure allows the resulting system to have a single, unified code base of pipelined operators, instead of a CPU-family and a GPU-family of vectorized ones. Lastly, the work of Menon et al. [22] to introduce SIMD vectorization in a CPU-based JIT engine is compatible with our design.

The compiler (sometimes) knows better. Writing code to be executed on a GPU can be a very subtle process [8, 6, 9, 15]. Conventional knowledge has it that a developer needs to explicitly reason about numerous low-level details, such as, among others, i) the organization of GPU threads in thread blocks, and of thread blocks in grids, ii) thread divergence within a thread warp, and iii) avoiding atomic operations. When this source of complexity is coupled with the complexity of implementing a code-generating engine, the end result can be very burdening to a developer. An observation that we made during the coupling of HetExchange with Proteus, however, is that the CUDA compiler has been becoming significantly better in optimizing code that has not been meticulously fine-tuned to the device-specific "magic numbers" required for thread block size, etc., to the degree that a lot of the conventional GPU coding wisdom [8] has become obsolete for modern GPUs. Thus, explicitly choosing to offload a part of the GPU code optimization complexity to the CUDA compiler reduces developer effort and focusing on the bigger system picture instead of on micro-optimizations.

8. REFERENCES

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

[2] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The case for heterogeneous htap. In *CIDR*, 2017.

[3] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230. ACM, 2018.

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

[5] S. Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

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

[7] S. Breß, B. Köcher, H. Funke, T. Rabl, and V. Markl. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *CoRR*, abs/1709.00700, 2017.

[8] J. Cheng, M. Grossman, and T. McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[9] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *SIGMOD*, pages 1603–1618, 2018.

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

[11] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*. IEEE Computer Society, 1993.

[12] K. O. W. Group. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf.

[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *TODS*, 34(4):21:1–21:39, 2009.

[14] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[15] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[16] T. Karnagel, D. Habich, and W. Lehner. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB*, 10(7):733–744, 2017.

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

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

[19] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment*, 11(13):2209–2222, 2018.

[20] C. Lameter. An overview of non-uniform memory access. *Communications of the ACM*, 56(9):59–54, 2013.

[21] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[22] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[23] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[24] NVIDIA. Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[25] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, pages 237–252, 2009.

[26] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

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

[28] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.

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

[30] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM, 2011.

[31] 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*, pages 417–432. ACM, 2017.

[32] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*. ACM, 2016.

[33] H. Wu, G. F. Diamos, S. Cadambi, and S. Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *MICRO*, pages 107–118, 2012.

[34] 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.