ADAMANT: A Query Executor with Plug-In Interfaces for Easy Co-processor Integration

Bala Gurumurthy*, David Broneske†, Gabriel Campero Durand*, Thilo Pionteck‡ and, Gunter Saake*
*Databases and Software Engineering
University of Magdeburg, Germany
†German Center for Higher Education Research and Science Studies
Hannover, Germany
‡Hardware-Oriented Technical Computer Science
University of Magdeburg, Germany

***Abstract*—Today's processor landscape is increasingly heterogeneous with the availability of co-processors. This landscape impacts query engines, as they need to be reworked to keep competitive performance by leveraging the underlying architectures. Such a rework might be costly if, for each external processor or SDK, peripheral components needed to be developed as well; resulting in redundant effort and adoption difficulties. In this paper, we propose an approach to overcome these shortcomings through ADAMANT – a query executor equipped with interfaces to plug-in new co-processors without reworking other components of a query engine. ADAMANT consists of 1) pluggable interfaces that allow interaction with co-processors, encapsulating operator implementations, and 2) a unified runtime that handles the execution on arbitrary co-processors, with a chunked execution model for scalable query processing. To evaluate ADAMANT's versatility, we plug different implementations of a CPU/GPU-based system (using OpenCL, OpenMP, & CUDA) and analyze their performance on TPC-H queries. We identify a 4x performance difference between an arbitrary chunked execution vs. a more architecturally conscious pipelined execution. Furthermore, our comparisons with HeavyDB show complex performance variations from speed-ups up to a factor of 2x from our hardware-conscious execution. We envision initiatives like ADAMANT to ease the study of complex optimizations required in co-processor systems, paving the way for efficient and portable data management tools without cutbacks.**

***Index Terms*—Co-processor acceleration, Hardware-aware query engine, Cross-device query execution**

I. Introduction

Today's hardware landscape is broad and diverse. Numerous co-processors differ in architecture, programming approach, and strengths, just to name a few. Applications such as database engines, running over such co-processors must be adapted to the underlying architecture, for efficient execution [42]. In particular, co-processor accelerated databases attempt to leverage the performance gains from different co-processor types in query processing [12]. The early 2000s saw increased use of GPUs for query processing*: Ranging from offloading a few database operators [26] to a fully-fledged query processor [4], [30], [31] – GPUs have wide support in today's DBMSs. With more and more SDKs being developed for GPUs, query engines over GPUs are also becoming more efficient. Currently, we see this trend branching towards other co-processors as well. New generation CPUs with SIMD acceleration [15], [29], [45], [49] and FPGAs with query execution in line-rate, and most importantly with good SDK support for these co-processors (OpenCL, OneAPI, CUDA, Verilog, etc.), has renewed the interest in co-processor acceleration for query execution [6]. However, this also comes with a new challenge, which is the frequent adaptation of query executors† for the different SDKs (on top of the co-processors) [16]. With various combinations of co-processors and SDKs, it is often the case that engineers have to develop multiple versions of query executors. This is challenging, as with every SDK, the existing query executor has to be updated.

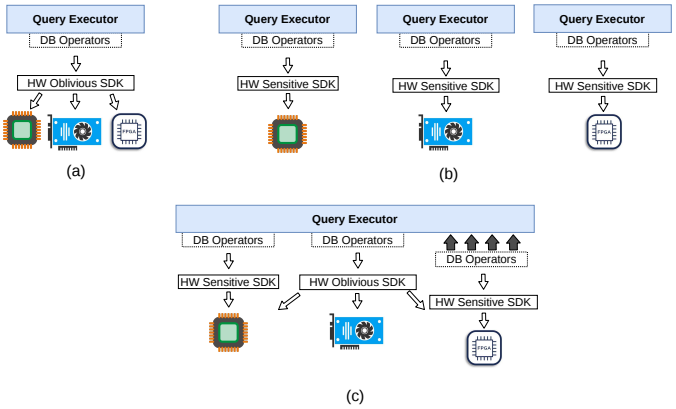

Figure 1: Three ways of architecting a query executor over co-processors [31]: a) using a common executor with a hardware-oblivious SDK; b) using a custom executor with a hardware-aware SDK per co-processor; c) a common pluggable executor for any type of SDK.

In general, a query executor can be extended with co-processors in two ways [31]: 1) using hardware-oblivious SDKs (cf. Figure 1(a)) or 2) using hardware-aware SDKs (cf. Figure 1(b)). The former method is portable to any of the SDK-supported co-processors but comes with the price

*DBMS with GPU support: https://dbdb.io/browse?hardware-acceleration=gpu

†We define the query executor as the component handling the execution of a query, which runs device-specific kernels and is responsible for supplying data to the processing devices.

of poor performance portability [47]. The latter case allows performance portability with the price of re-working the query executor. As the co-processor landscape gets broader with different accelerators, the disadvantages of these approaches are ever more noticeable. Therefore, we need a query executor that is extensible while not compromising performance. To this end, we propose a unified query executor – ADAMANT – that supports the free plug-and-play use of any SDK/co-processor, without reworking the execution modules (see Figure 1(c)).

To realize the functionality of ADAMANT, our executor is split into two parts: a set of device-pluggable interfaces and a unified runtime. These parts address two key challenges. 1) **Multiple implementation alternatives:** With multiple SDKs per co-processor (e.g., a GPU has OpenCL, CUDA, oneAPI, etc.), one has to capture multiple versions of database operations. In addition, there may be multiple versions for an algorithm with a single SDK specialized for a specific workload [52]. Here, the device-pluggable interfaces define signatures for database operators so that any new implementation can be plugged into the system. Using the interfaces, we can freely couple any SDK together with its operator implementation. 2) **Handling co-processor execution:** With each co-processor, a runtime has to manage the data transfer across the device, as well as the execution itself. These functionalities are highly SDK dependent, such that updating SDK calls for one device might affect the functionalities for another. We overcome this challenge with a unified runtime that supports abstract execution models. These models handle query execution over any co-processor that is plugged in. Additionally, the models support `larger-than-memory` data sizes, i.e., processing data that does not fit completely in the co-processor memory. we implement a chunked execution model to transfer data without putting pressure on device memory.

The novelty of our approach can be considered by seeing existing related approaches that support co-processor acceleration with varying degrees of extensibility [3], [25]. Even though these systems support execution over heterogeneous processors (providing scheduling, data placement, etc.), their support for query execution is limited. Unlike these, our unified runtime expresses execution models that support query execution on arbitrary co-processors.

Overall, in this paper, we architect a pluggable query engine to couple a new co-processor or API with an existing co-processor without reworking the complete query engine.

Our main contributions are as follows.

- A query executor that allows easy integration of co-processors.
- Alternative execution models for co-processor acceleration, which implement operator-at-a-time, chunked execution, pipelined execution, and 4-phase pipelined execution.
- An experimental study with two devices (CPU, GPU) and three different API implementations (OpenCL, OpenMP, and CUDA) showing the versatility as well as shortcomings of our query executor.

The remainder of this paper is structured as follows. First, we present the background for co-processor accelerated query processing (Section II). In Section III, we present the different tiers in our query execution engine and list available interfaces that enable the pluggability of co-processors. Next, in Section IV, we explain the different execution models incorporated in the runtime. Section V covers the details of our experimental study of ADAMANT, with various heterogeneous operator implementations. Finally, in Section VI, we provide further context to our study by reviewing related work, and we conclude our work in Section VII.

II. Co-processors for Query Execution

Today, co-processors are deployed across various domains (e.g., GPUs are used for gaming, data mining, deep learning, etc.) [36], [40]. As a consequence, there is a steady rise of SDK alternatives, as well as libraries, for co-processors [14]. In this section, We briefly explore the coprocessors and their SDKs in the context of query execution.

A. Diversity in Co-Processors

To overcome design limitations of CPUs[‡], single database operators or even complete queries are commonly offloaded to co-processors. Some of the commonly used co-processors are:

- **MIC (Many Integrated Cores):** This device has many CPU cores with multiple levels of shared cache lines. It supports operator pipelining and data parallel execution [44].
- **GPU (Graphical Processing Unit):** GPUs have thousands of light-weight cores with SIMT-style execution. They are suitable for data parallel execution of an individual database operator or a complete query [12].
- **FPGA (Field Programmable Gate Array):** FPGAs are custom-programmable logic devices. Given enough resources, an FPGA system designer can build a deep pipeline of database operators or replicate these for data parallelism [20], [23].

All these devices vary in their architectures; therefore, suitable programming abstractions are built for them. Below, we discuss some of these abstractions and the levels of expressiveness available in them.

B. Diversity in Programming Abstractions - SDKs

Co-processor SDKs give access to specialized hardware components. For example, Intel's SSE instruction set[§] gives access to SIMD features of CPUs. Based on SDKs, we identify three access levels while integrating a query engine with a co-processor. At the lowest level, vendor-specified SDKs give access to almost all hardware components of the co-processor. As a result, they offer the best performance at the cost of

[‡]Like the power wall problem from Dennard scaling breakdown, where the increase in transistor size leads to power leaks in the form of heat, resulting in a poor power-to-performance ratio.

[§]https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

poor code portability [34]. Next, there are wrappers that cover SDKs, abstracting device-specific details (e.g., OpenCL). In addition, they have standard functional abstractions for all supported hardware. Finally, co-processors have libraries with pre-written functions. These are written by device experts using one of the low-level SDKs, abstracting all key implementation details from an end user (e.g., OpenBLAS, cuDNN) [52].

Now that we have seen the diversity in SDKs and co-processors, in upcoming sections, we will explain our coprocessor pluggable query executor and its necessary components.

III. A Query Executor to Plug-in Co-Processors

Since a host CPU usually handles the execution routines of a co-processor, our architecture (depicted in Figure 2) also has the unified runtime running on a host. The runtime interacts with the plugged co-processor using predefined device interfaces. These interfaces act as functional boundaries, separating the query engine from co-processor SDKs. Finally, we introduce an intermediate task layer to handle alternative implementations of a database operator across these SDKs. Overall, our architecture is split into three loosely coupled layers. The layers and their responsibilities are:

- The **Device Layer** represents the implementation of the driver on the target.
- The **Task Layer** links runtime handlers to database operators on the underlying device driver.
- The **Runtime Layer** acts as the host handling the execution across multiple devices.

As shown in Figure 2, our runtime takes a query plan (generated from any existing optimizer) translated into a primitive graph with annotations marking the target device. Using these annotations, the custom execution models at the runtime layer (see Section IV) process the primitives using the interfaces in the device and task layers, respectively. Thus, our executor is split into a host-dependent runtime layer interacting with a flexible co-processor-dependent device and task layers. Below, we first explain these interfaces in detail and show how to construct arbitrary execution models for co-processor acceleration.

A. Device Layer

There are multiple SDKs per device, each providing varying performance benefits [38]. For example, profiling the bandwidth range of OpenCL and CUDA in Figure 3 shows variations in transfer bandwidths. Generally, results show a lower bandwidth range for OpenCL compared to CUDA. This difference arises from OpenCL's translation overhead. Such a minor yet significant difference affects the overall query execution considerably. Similar performance differences can also be observed in other functionalities of the wrappers (e.g., during kernel launch, memory allocation, etc.). Therefore, it is expected that if a newer and more efficient SDK is available, these functions will have to be rewritten. Therefore, for such re-work, we propose a device layer, which we use to pack SDK functions into two groups: 1) kernel management and 2) data management. We split the kernel functions separately and make them optional, as not all the SDKs support runtime compilation of kernels. On the contrary, the interface functions for data management are mandatory to be able to plug in a co-processor, as data management needs to be handled explicitly at runtime. These data management tasks include – allocation/freeing memory space in the device and transferring data into the allocated space.

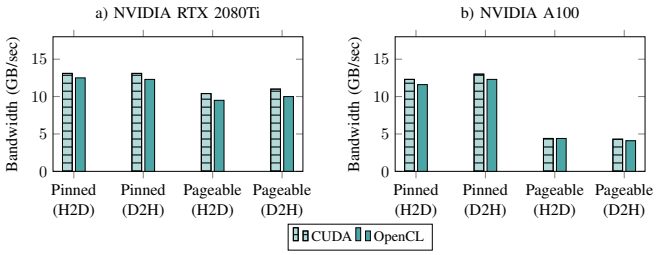

Figure 3: Data transfer bandwidths using CUDA and OpenCL across GPUs. H2D: Host to device, D2H: Device to host

Furthermore, we dedicate an interface to explicitly transform data from one SDK's data type to another. To understand data transformation complexity, consider a GPU using libraries – Thrust & Boost.compute and SDKs – CUDA & OpenCL. Each interprets a GPU's memory space in its own data type, as shown in Figure 4.

Normally a host is unaware of the relation between SDKs. Therefore, any data is transferred into the host first, transformed into the target data format, and transferred back into the device. Such unwanted transfers to and from a co-processor

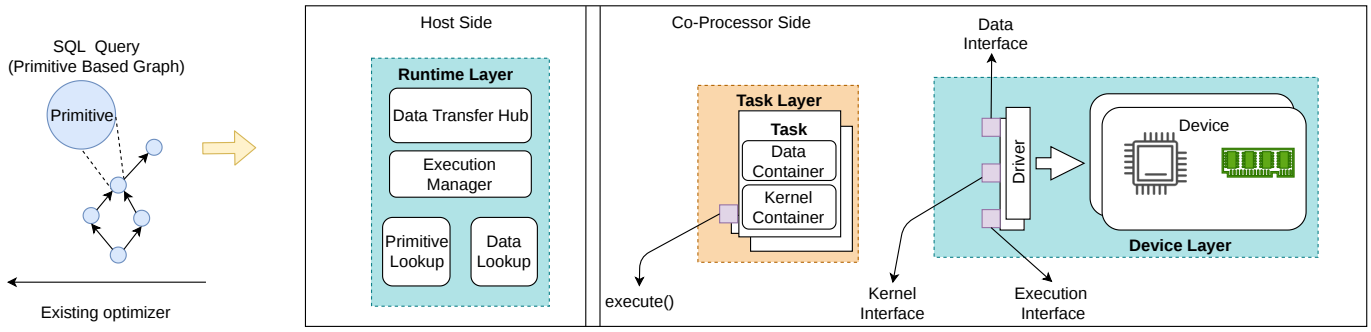

Figure 2: Architecture with a unified runtime and interfaces (purple blocks) to interact with plugged components.

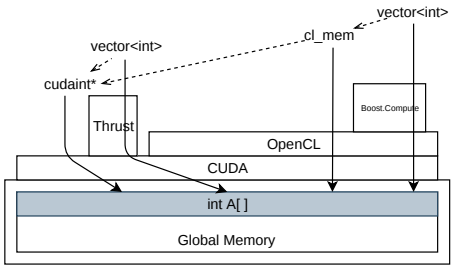

Figure 4: Data types across SDKs. Data type (Solid) used by a developer & in SDKs (dotted)

can be avoided with a transform function, which internally transforms the memory objects from one representation to another without actual data movement. Based on the criteria, we have defined ten interfaces for the device layer:

place_data(data, size, offset) - push data to the device.
retrieve_data(id, size, offset) - receive data from the device.
prepare_memory(size) - allocate memory in the device.
transform_memory(source, target) - convert data type from source to target.
delete_memory (id) - de-allocates memory in the device.
prepare_kernel(name, location) - compile functions in the device.
initialize() - set relevant properties for the co-processor.
create_chunk(ID, chunk size, offset) - access a subset of data in the device.
add_pinned_memory(ID, chunk size, offset) - reserve host-accessible memory.
execute() - execute any task tagged to the device.

With any new SDK developed for an existing or new co-processor, one can realize these interfaces to couple them with our runtime. Next, let us see a sample integration of an OpenCL-programmed GPU into the ADAMANT system.

1) Case Study - Integrating a GPU: To showcase the usage of our ADAMANT, we take the OpenCL GPU wrapper as a case study integrating it into our ADAMANT system. Similar integration can be done for other GPU wrappers as well as other systems.

Initially, a column has to be placed into the device memory for execution. We realize this operation with the `place_data()` interface, which takes an input array, its size, and optionally the starting index of the data. The corresponding OpenCL wrapper code looks like the one given in Listing 1, where we show the functions used for buffer creation and memory transfer.

```
int OpenCLDevice::place_data(unsigned int* data, size_t size,
      size_t start_idx ) {
...
 _m_data_buffer = clCreateBuffer (_m_context, NULL, (size) * sizeof
       (int), NULL,&_m_err);
 _m_err = clEnqueueWriteBuffer(_m_device_queue, _m_data_buffer,
       CL_TRUE, 0, size * sizeof(T),data, 0, NULL, NULL);
... }
```

Listing 1: OpenCL code for transferring data to a GPU

Additionally, one can include support for unified memory if possible. In the case of GPU supporting unified memory, it is added in the `add_pinned_memory()` as shown in Listing 2.

```
int OpenCLDevice::add_pinned_memory(short alias, size_t size,
      size_t start_idx ) {
...
 _m_data_buffer = clCreateBuffer (_m_context,
       CL_MEM_ALLOC_HOST_PTR, (_size) * _m * sizeof(T),
       NULL, &_m_err);
...
}
```

Listing 2: OpenCL code to allocate space in unified memory

We explicitly define these pinned memory functions to take advantage of fast data transfer. We use this memory space to transfer chunks onto the device while utilizing the dedicated memory to store intermediate results. More details on using the pinned memory are given in Section IV. Finally, we clear these allocated memory spaces as shown in Listing 3.

```
int OpenCLDevice::delete_data(short alias ) {
...
 cl_int err = clReleaseMemObject(_m_data_buffer);
...
}
```

Listing 3: OpenCL code to delete space

Now that the data management functions are present, we focus on integrating the kernel compiler and its corresponding execution functions. First, we compile a primitive kernel as shown in Listing 4.

```
int OpenCLDevice::prepare_kernel(short alias, string _kernelSrc) {
...
 cl_program _m_program = clCreateProgramWithSource(_context, 1, &
       _kernelSrc, NULL, NULL);
cl_int _m_err = clBuildProgram(_m_program, 1, &_device, _cmdAgrs.
      c_str(), NULL, NULL);
_m_kernel = clCreateKernel(_m_program, _kernelName, &_m_err);
...
}
```

Listing 4: OpenCL code to compile a kernel

Our system compiles all the pre-existing kernels during initialization. These compiled binaries are tagged to their corresponding tasks (detailed in the next section), therefore, `task->execute()` performs the current task. This `execute()` in OpenCL can be implemented as in Listing 5.

```
int OpenCLDevice::execute() {
for (int i = 0; i < _m_args_size; i++)
_m_err |= clSetKernelArg((*_m_iter).second, i, sizeof(cl_mem), &
       _m_argument_buffer);

for (int i = 0; i < _m_param_size; i++)
_m_err |= clSetKernelArg((*_m_iter).second, i + _m_args_size,
       sizeof(int), &_param[i]);

_m_err = clEnqueueNDRangeKernel(_m_device_queue, _kernel, wd,
       NULL, &_globalSize, &_localSize, 0, NULL, NULL);
}
```

Listing 5: OpenCL code for kernel execution

2) Integration of Other Co-Processors: Other than GPUs, we can also integrate FPGAs and other co-processors into our system. For the case of FPGA, we can consider generating binary files from the input for transferring into the device as `place_data()` and reading back from binary to be `retrieve_data()`. Since FPGAs are commonly used to execute operations directly as soon as the data arrives on the device, the DMA function for data transfer will act as `execute()`. However, this `execute()` must be capable of targeting the right task. This depends on the implementation of these tasks. In the case of sophisticated mechanisms such as creating a runtime configurable overlay [7], [21], the device driver must be capable of handling the execution.

Limitations: One of the caveats of our system is the integration of near-data processors such as smart NICs that sits between a host and a coprocessor/data store. Still, one can support such smart NICs extending the `execute()` interface. Here, the custom driver for `execute()` must be capable of differentiating between the NIC and the target executing the operators individually in them. Here, the smart NIC and the coprocessor are represented as a single co-processing entity in our ADAMANT system. Thus, our ADAMANT can be pluggable with any new device wrapper as well as a coprocessor without any changes to other execution system components.

B. Task Layer

The task layer encapsulates multiple implementations of a database primitive. Once again, their performance varies according to the implementation. For example, a straightforward map and reduce will vary in their performance based on the SDK in which they are implemented. In fact, OpenCL (represented with a circular mark in the graph in Figure 5) and device-aware implementations (CUDA, OpenMP) show mostly the same performance. However, more complex operations will have clear variations in their performance (see Section V).

Apart from the results in the experiments, the implementation approaches can be: 1) hand-written, 2) library, or 3) generated on the fly (compiled) during runtime. Therefore, this layer collects implementations (task model) and enforces functional signatures of database operations (primitive definitions).

1) Task Model: As discussed above, a task reduces the complexity of including a new implementation variant without changing the device interface. We propose using two containers to handle the execution of a task or even a series of tasks.
1) `Kernel Container`: This is a simple adapter with additional runtime information required for executing a custom-written function. In the case of runtime compilation, the kernel string or generator is present in the container.
2) `Data Container`: Manages data formats for a task. Internally, a lookup table is used for data transformations. Using these, our runtime can handle data transfer and execution on different devices. With this generic task model, we can introduce database-specific operations defining the signatures of individual database primitives.

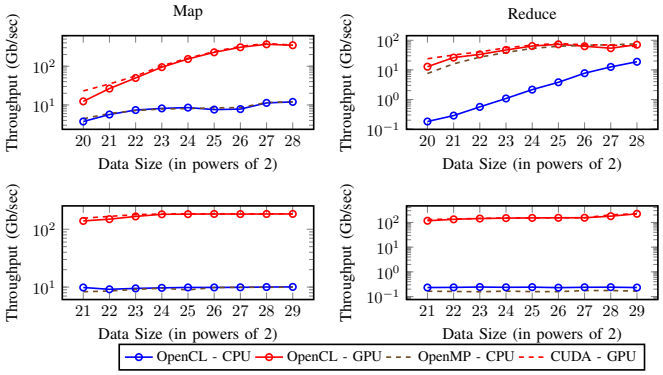

Figure 5: Performance of map & reduce depends on the underlying implementation, as well as the device. (The results are measured on top: NVIDIA RTX 2080Ti and Intel core i7-8700 & bottom: NVIDIA A100 and Intel Xeon Gold 5220R).

2) Primitive Definitions: Primitives are granular functions that build a database operator. We identify the common primitives surveying related work [13], [28], [30], [43]. Additionally, we also define the I/O signatures for these primitives as detailed in Table I. Therefore, any custom implementation of primitives can be included in the system, given that they adhere to the I/O semantics. This also helps include varying implementations of a primitive in our ADAMANT system. Further, with our I/O semantics, we can freely combine primitive implementations from different wrappers: like an OpenCL implementation of arithmetic followed by a reduce implemented using CUDA for a single device.

Query Pipelines: Apart from the awareness of primitive signatures, our system is also aware of the characteristics of these primitives. Specifically, ADAMANT is aware of pipeline breakers (denoted with †in I) and materializes their intermediate results into the device memory. These pipeline breakers mark the end of a query pipeline. Thus, given a query with several pipeline breakers (for example, Q3 of TPCH), our system splits it into its pipelines. These pipelines are considered an execution group, and all primitives are executed together (more details on execution are given in Section IV). Since a query is processed pipeline-wise, our framework can also work with compiled operators as they are also forced to generate code until a pipeline breaker before the next operator in the query can be executed [13].

3) I/O Definitions: Apart from the primitive functional definitions, we explicitly define I/O definitions to call an appropriate primitive further down the execution pipeline. For example, a selection primitive might return bitmaps or even a position list instead of column values to reduce the transfer load. However, if the materialization primitive is unaware of the incoming data scheme, it might generate wrong results (or even run into system exceptions). Therefore, we encode some of the standard I/O semantics of the primitives mentioned above in the data edges. Hence, when a selection produces its results as a bitmap, the corresponding materialize can be executed. Moreover, the result of a primitive might be forwarded to different primitives in the plan. For example,

Primitive definition	Description
MAP(NUMERIC in[n],NUMERIC out[n])	Does one-to-one mapping operation e.g. arithmetic operation.
AGG_BLOCK(NUMERIC in[n],(NUMERIC out)†	Does reduce operation on input (in) into result space - out.
HASH_AGG(NUMERIC in1[n],NUMERIC in2[n],HASH_TABLE hashTable[m])†	Does group-by aggregation of in2 based on groups in in1. In case of *COUNT* in2[n] is not required.
HASH_BUILD(NUMERIC in[n],HASH_TABLE hashTable[m])†	Populates the hashTable with the input - in.
HASH_PROBE(NUMERIC in[n],HASH_TABLE hashTable[m],JOINLEFT left[n],JOINRIGHT right[n])	Returns joins pairs in left and right respectively based on input in probing over hashTable.
SORT_AGG(NUMERIC in[n],PREFIX_SUM pxsum[n],NUMERIC aggregates[m])†	Does group-by aggregation over sorted data, with positions pointed by computing their prefix-sum.
FILTER_BITMAP(NUMERIC in[n],BITMAP bitmap[k],NUMERIC parameter)	Filters input in based on the parameter mentioned and stores the results in form of bitmap. Here, k = n/b - where b is the size of the bits packed per unit data.
FILTER_POSITION(NUMERIC in[n],POSITION position[k],NUMERIC parameter)	Filters similarly like FILTER BITMAP, but returns the position of selected input. The size of the result is estimated.
PREFIX_SUM(NUMERIC in[n],PREFIX_SUM pxsum[n])†	Computes prefix sum for a sequence of sorted input or input with series of 1s and 0s.
MATERIALIZE(NUMERIC in[n],BITMAP bitmap[k],NUMERIC output[m])	Returns the column values in input based on the bitmap.
MATERIALIZE_POSITION(NUMERIC in[n],POSITION position[k],NUMERIC output[m])	Returns the column values in input based on the position list.

Table I: Primitive definitions for encapsulating multiple database operator implementations

a hash join's results can be materialized separately for the left or right table accordingly. Based on these scenarios, we define the following I/O semantics:

- **NUMERIC** - Any numeric or column values.
- **BITMAP** - A bit-packed result. These are the results of a FILTER primitive.
- **POSITION** - A position list. These are the results of a FILTER primitive.
- **PREFIX_SUM** - results of PREFIX_SUM primitive. Useful with SORT_AGG.
- **HASH_TABLE** - result of HASH_BUILD or HASH_AGG.
- **GENERIC** - Any custom data semantic (e.g., a specialized tree structure for filtering).

Using these three components, one can form a query execution plan using primitives. This enables a developer to include any custom implementation into the runtime and query execution.

C. Runtime Layer

The runtime layer interprets a query execution plan, executing it in the target devices. Below are the components present in the layer.

Primitive Graph: It models a query execution plan with primitives (Section III-B2) as nodes and the data flow across primitives as edges. The graph encodes additional data information at the edges. Below are some details on the information encoded:

- **data ID** - unique ID for the data path.
- **device ID** - data location across devices.
- **processed until** - index until which the data have been processed so far.
- **fetched until** - index until the input is transferred into device memory.

Using the data ID and device ID, we infer the type of transfer necessary for the target device. Pointers defined as processed & fetched-until allow for parallelism in query execution.

Data Transfer Hub: The data transfer hub has three main tasks:

1) **load_data()**: loads data to the target device before execution. This includes either loading the complete data into the device or incurring overhead from partial loads. Internally, this function calls `place_data()` to load the input.
2) **router()**: Handles all SDK-to-SDK and device-to-device data transfers. This function iterates over all the incoming edges to a primitive and loads the data to the target device. Internally, the function calls the interfaces: `place_data()`, `retrieve_data()`, and `transform_memory()`.
3) **prepare_output_buffer()**: It estimates and creates a result space for a given primitive. It also handles data semantics based on the primitive.

In summary, the runtime couples operator implementation with device interfaces. Thus, the three layers enable query execution of any plugged co-processors. Such an out-of-the-box query execution is possible because of multiple execution models present in the runtime. In the next section, we describe these execution models for plugging devices.

IV. Execution Model Alternatives for Co-Processors

Execution models directly define the process and data flow during query execution. For co-processor acceleration, an execution model defines the amount of memory to be used in a co-processor and the execution flow within a co-processor as well as across the co-processor and its host. Defining a

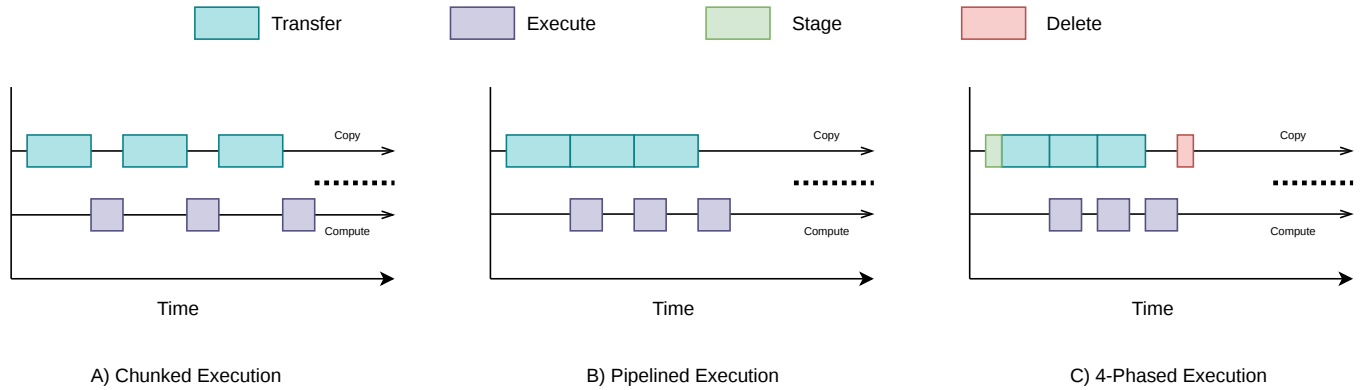

Figure 6: Execution model alternatives for co-processor acceleration

suitable execution model for co-processor acceleration in turn characterizes the execution flow of our runtime.

In this section, we explore the scalability limitation in the operator-at-a-time execution model with examples. To overcome the limitation, we explain an abstract chunk-based execution model that can support any arbitrary co-processor. Next, we modify this chunked execution model to be hardware-aware using GPU as a case study.

A. Limitations in Operator-At-A-Time Execution in Co-Processors

Operator-at-a-time (OAAT) is one of the common execution models for query execution in co-processors. The model stores table data within device memory to avoid costly transfers. The direct dependency on the memory capacity makes the model not scale with growing data sizes. To illustrate this, we plot the data size of the input for different queries in the TPCH benchmark and the complete TPCH dataset against the memory capacity on various GPUs (cf. Figure 7-left). From the results, we can observe that only some TPCH queries can be executed on a device with input data completely in the device memory. As a normal OLAP query requires only a few columns from the complete dataset, storing the complete dataset reduces the space to store intermediate results. For example, the query plan for TPCH Query 6 in Figure 7-middle has the memory footprint shown in Figure 7-right during execution. Thus, storing the complete dataset in the co-processor memory reduces the space available for intermediate results of a query. To reduce such memory pressure on the co-processor, we need an alternative execution model.

As an alternative to the operator-at-a-time, a scalable chunked execution model is already available [24]. We construct a similar chunked execution model using the interfaces discussed in the previous sections. Using this execution model, our runtime can scale query execution over any arbitrary co-processor.

B. Chunked Execution for Arbitrary Co-Processors

Even with chunked execution, a long query execution plan might generate multiple intermediate results utilizing the complete memory space of a co-processor. Therefore, our execution plan executes a query pipeline-wise to reduce both the memory load, as well as processing load in the device.

Algorithm 1: Chunked execution

```
foreach chunk C of input do
    foreach Primitive P in pipeline (QEP) do
        Edge ie = incoming_edges(dag[P]);
        router(ie.source_device_ID,
          ie.target_device_ID,c,chunk_size);
        available_device[target_device_ID]→
          prepare_memory(output_size);
        available_device[target_device_ID]→ execute();
```

The chunked execution constructed using our interfaces is given in Algorithm 1. The execution starts with a chunk of input transferred to the co-processor. This chunk is processed through a complete pipeline and the intermediate result of the final pipeline operator is persisted, while others are overwritten by the results of processing the next chunk. The overall memory consumed for intermediate results depends on the chunk size; therefore, only a fraction of the memory is utilized. Since a chunk has to be processed until the end of a pipeline, the next chunk is transferred only when the current chunk is processed. Here, the transfer waits for the execution to complete before transferring the next chunk. Even though the execution model works with arbitrary data sizes, its performance might not be optimal due to constant data transfers. Such transfer delays are hardware-dependent and, therefore, can be improved only using a hardware-aware approach.

Since improving chunked execution is hardware-centric, we take a current generation GPU as a case study and utilize its components in improving our execution model.

C. Case Study: Pipelined Execution in GPUs for Concurrent Execution with Data Transfer

Since data transfer is a bottleneck, we hide the transfer time with concurrent execution (cf. Figure 6-b). However, the transfer delay is so high that hiding it with a single primitive execution will not be beneficial. Hence, we hide the transfer of a data chunk with the execution of a complete pipeline. We incorporate this `copy-compute` routine into our runtime

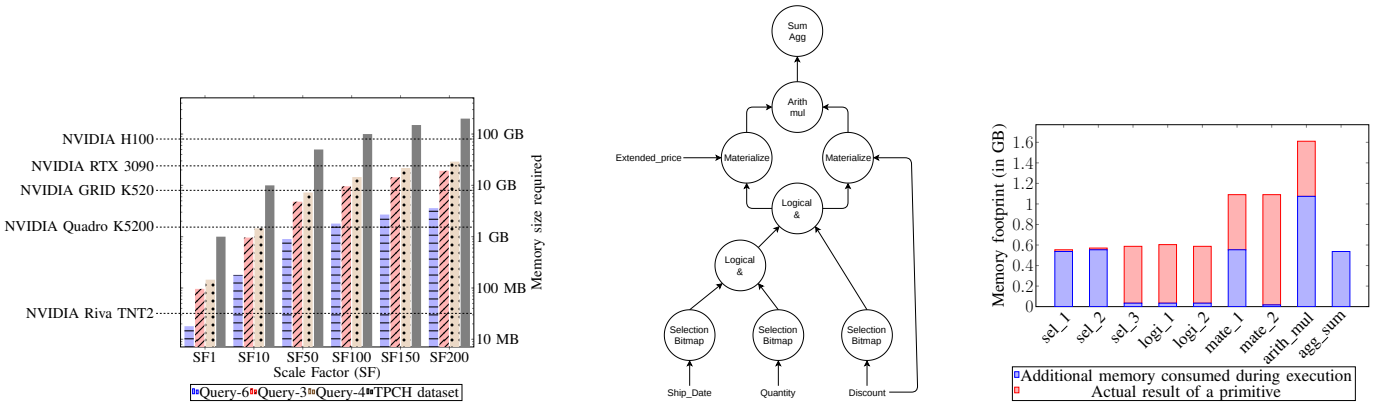

Figure 7: Limitations in operator-at-a-time execution. Left: Memory capacity in GPUs vs. memory required for TPCH dataset. Middle: Query execution plan for TPCH-Query 6. Right: Memory footprint for Query 6 execution with SF=100.

Algorithm 2: Pipelined execution

```
foreach Primitive P in pipeline(QEP) do
    thread transfer = spawn_thread(transfer_data() );
    foreach i=0 until input/chunk do
        Edge ie = incoming_edges(dag[P]);
        wait_until(ie.fetched_until ≤ ie.processed_until);
          available_device[target_device_ID]→ execute();
        ie.processed_until+=chunk_size
transfer_data(){
foreach Chunk C of input do
    foreach Edge ie in incoming_edges(dag[P]) do
        router(ie.source_device_ID,
          ie.target_device_ID,c,chunk_size);
        available_device[target_device_ID]→
          prepare_memory(output_size);
        ie.fetched_until+=chunk_size;
    }
```

using separate threads for data transfer and pipeline execution on a co-processor.

We track the processed chunks to synchronize the threads effectively. The execution thread keeps track of the amount of data processed using a counter `processed_until`. A similar counter - `fetched_until` - is used to track data transferred so far. If the fetched_until value is smaller than processed_until, the execution thread waits for the data to be transferred by the transfer thread, and vice versa. The threads also synchronize at the end of each pipeline breaker and start with the next pipeline. Varying this chunk size, we can balance transfer and execution speed. In case the transfer is faster, we can push more data into the device.

Even though the execution model hides execution with the transfer, the overall transfer overhead is still the same as that of naive chunked execution. We improve this further using a four-phase approach, described in the next section.

4-Phase Pipelined Execution With Memory Reuse: Since transfer is a severe bottleneck, improving it should increase performance. We optimize the transfer delay in this execution model, as shown in Figure 6-c, with four different phases.

The detailed execution of the four-phase query execution is given in Algorithm 3. As the memory transfers for a query are predefined, we use *pinned memory* (cf. Figure 3) for faster transfer. As pinned memory is accessible to both the host and the co-processor, the host directly transfers data here while the co-processor executes its primitives. One problem with copy-compute is that the copy phase might overwrite the currently executed data. However, we ensure the data is not overwritten using the `processed-until` index.

Algorithm 3: 4-phase pipelined execution

```
//Stage Phase
foreach Primitive P in pipeline(QEP) do
    foreach edge E in P do
        if Source(E) is Data_Scan then
            alias1 = available_device[target_device_ID]
              →add_pinned_memory(memory_size);
            data_dictionary.insert(E,alias1);
              alias2=available_device[target_device_ID]
              →add_pinned_memory(memory_size);
            data_dictionary.insert(E,alias2);
        else
            alias=available_device[target_device_ID]
              →prepare_memory(memory_size);
              data_dictionary.insert(alias,E);
//Copy-compute phase
foreach Primitive P in pipeline(QEP) do
    thread transfer = spawn_thread(transfer_data() );
    foreach i=0 until input/chunk do
        Edge ie = incoming_edges(dag[P]);
        wait_until(ie.fetched_until ≤ ie.processed_until);
          available_device[target_device_ID]→ execute();
        ie.processed_until+=chunk_size
//Delete phase
foreach Primitive P in pipeline(QEP) do
    foreach entry in data_dictionary do
        available_device[target_device_ID]
          →delete_data(entry.alias);
```

To avoid such scenarios, we create two identical memory spaces to alternate execution and transfer. The transfer and execution threads alternate between these memories that access the chunks, as shown in Figure 8. Additionally, the intermediate results of any pipeline breaker are also transferred back to

the host using pinned memory. All other intermediate results are stored in the device memory itself. Once the execution is complete, we deallocate these memory locations. In summary, execution starts by creating pinned memory spaces – stage phase – over which the chunks are copied – copy phase. Once a chunk is copied, the compute phase processes these data. Once all the chunks are processed, the deletion phase deallocates the memories for the next queries.

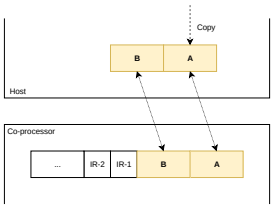

Figure 8: Dual memory spaces for concurrent transfer-execution

Replacing `add_pinned_memory()` with `add_data()`, we make the above execution models support any arbitrary co-processor. However, the performance still depends on the implementations of the device driver. To this end, we have developed custom drivers for GPU and CPU using CUDA and OpenMP, respectively, in addition to a standard implementation in OpenCL. We use these drivers to evaluate the performance of our ADAMANT framework, and the results are discussed in the next section.

V. Experiments

We evaluate in this section the performance of our custom primitive implementation, the overhead from our abstraction layers, and the performance of using the above-mentioned execution models, the latter with special consideration for larger-than-memory processing. For our evaluations, we use the device drivers: OpenCL (for CPU), OpenMP, OpenCL (for GPU), and CUDA, running on top of two different environments. Details about the environments are given in Table II. The primitives over the evaluated drivers follow semantically similar implementations. All these implementations are written in C++[¶]. For the baseline, we consider HeavyDB (formerly MapD) for its compiled execution model and compare it with our proposed execution models.

	Setup 1	**Setup 2**
CPU	Intel(R) Core(TM) i7-8700	Intel Xeon Gold 5220R
GPU	GeForce RTX 2080 Ti	Nvidia A100
GCC	9.3.0	8.4.0
OpenCL	2.1	2.1
CUDA	11.0	10.1
OS	Ubuntu 18.04	Ubuntu 20.04

Table II: Enviromental setup

[¶]code available at https://git.iti.cs.ovgu.de/dead-ops/ADAMANT/-/tree/master

A. Profiling Primitives

We profile the throughput of our primitives using 2^{28} integers values (1GB) in random distribution. In addition, our filter operator can perform early and late materialization [1] with bitmaps as an intermediate data type. Finally, we measure the performance of both approaches. Similarly, as the hash join has both the hash-build phase and the hash-probe phase, we measure them individually. We use linear probing as the underlying hashing technique. The hash table is placed in global memory, and all the threads compete to place their data in a bucket, which is resolved using atomic operations. The performance profile of these primitives is shown in Figure 9 for two hardware configurations.

Filter (Bitmap): The results in Figure 9 (a) are nearly constant for the different devices as we perform a bitwise comparison of the values in an array. Since each comparison of input takes roughly the same amount of time irrespective of their selection, the performance graph is similar to that of a map in these devices (cf. Figure 5). However, on both devices, OpenCL performs better than OpenMP on the CPU but is equivalent to CUDA on GPU. In our case, the OpenMP variant suffers from data movement overheads, as the hardware threads are explicitly scheduled, whereas OpenCL handles them internally. Perhaps further evaluation with varying thread sizes could improve this.

Filter (with Materialize): Comparing Figure 9 (a & b), we see that adding materialization leads to a significant performance drop in a GPU - about 30% the performance from using only bitmap. This is mainly from the time taken to extract bits from a bitmap in a GPU. Since we pack results from multiple inputs into a single bitmap, GPU threads must extract their respective bitmap input cooperatively, leading to performance degradation. However, such an impact of materialization is minimal for CPUs, as threads are scheduled with a sequence of 32 input values, avoiding data sharing among threads.

Hash Aggregation: Our hash aggregation uses a single global hash table for aggregation. Based on the results in Figure 9 (c), we see that the performance of OpenCL decreases drastically with increasing group sizes. As the data from SIMT threads is served through a common memory controller, inserting multiple data in parallel requires more time.

We also see the profile of CUDA is not deteriorating with larger group sizes than that of OpenCL. We believe this is due to the static scheduling of threads in OpenCL. Finally, the high performance of GPUs comes from their faster internal bandwidth from memory controllers.

Hash Build: Similar to hash aggregation, hash build also has a shared hash table for insertion. We see that the hash build performance drops with larger data sizes. This is mainly due to the repeated data insertion calls from threads for larger data sizes. On the other hand, the CPU performance is still the same. Again, the threads spawned in a workgroup lead to minor performance differences across OpenCL and OpenMP execution. Additionally, by comparing the performance with hash probing, we can easily identify insertion overhead using

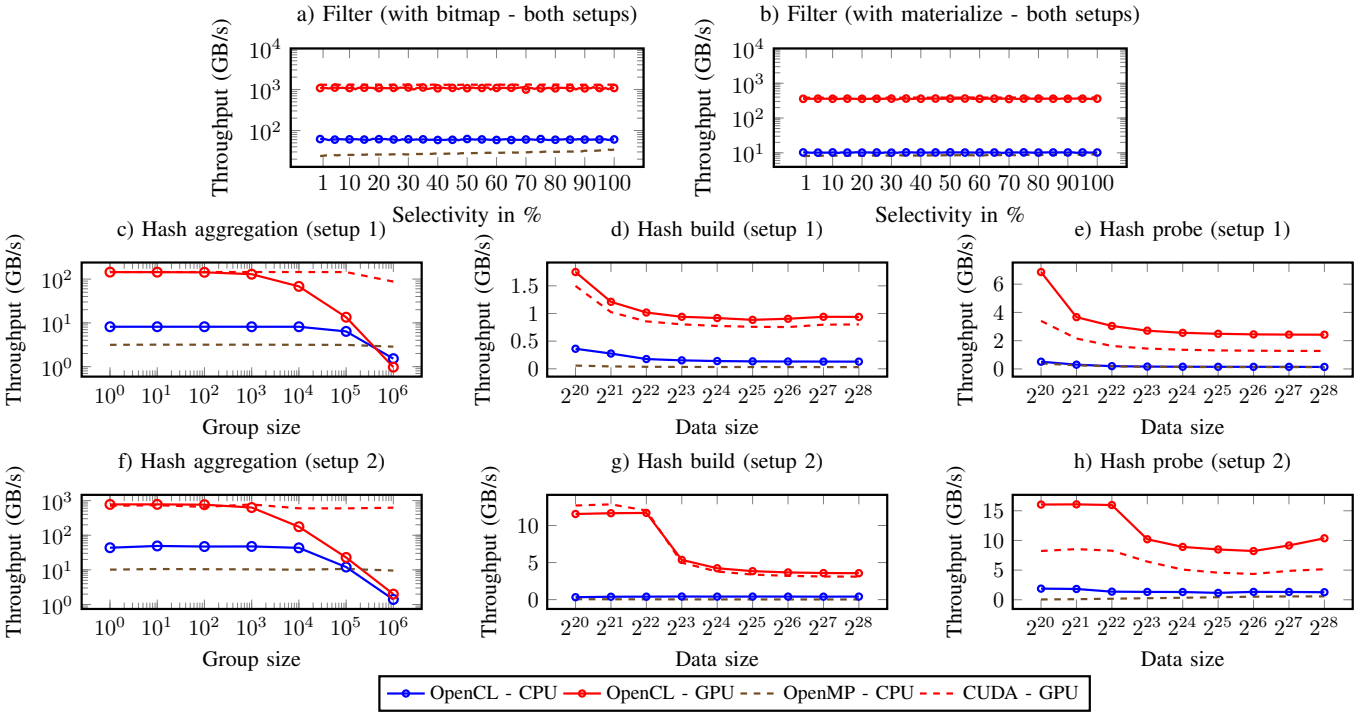

Figure 9: Profile of primitives in OpenCL, OpenMP and CUDA

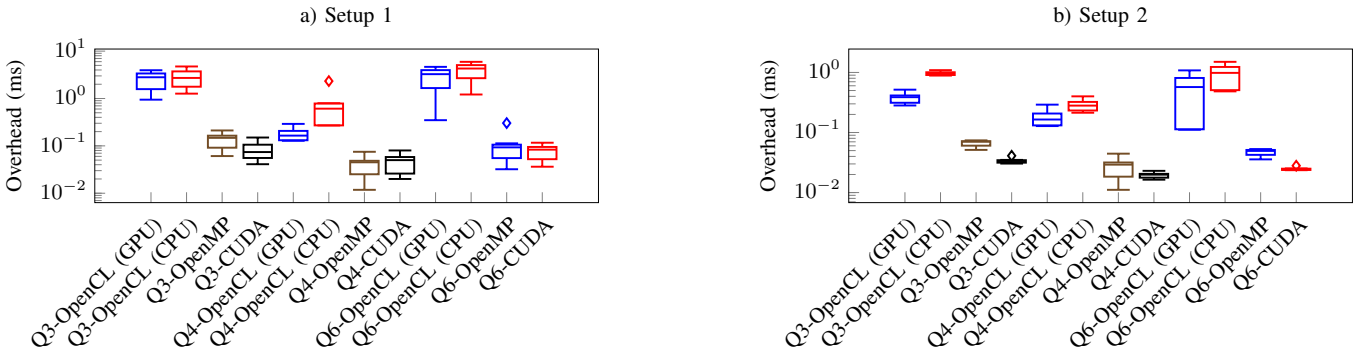

Figure 10: Overhead due to abstraction layers

a single shared hash table. Here, we use atomics for insertion, which serializes the threads during insertion. The difference in performance indicates this serialization overhead of atomics.

Hash Probe: Since hash probe follows nearly the same execution flow as the hash build, the reflected performance also has similar characteristics to a hash build. However, the performance from CUDA is affected by the probe, compared to OpenCL. This might be due to the influence of the order of threads accessing the global memory.

B. Impact of Abstraction Layers

As our abstraction layers are loosely coupled, they incur overhead in query execution. To understand this overhead, we measure the difference between the overall execution time and the total sum of processing time of the individual primitives of a query. The results in Figure 10 show a maximum overhead for OpenCL wrappers, compared to CUDA and OpenMP. This overhead arises from explicit data mapping to a target kernel, whereas OpenMP and CUDA do not need such data mapping explicitly. Based on the results, we see that the abstraction layers and the overhead of our execution model are minimal compared to direct execution. Furthermore, by comparing the performance across devices, we see that the hardware-sensitive implementation for the primitives plays a major role in performance. In general, our OpenCL implementation incurs significant delays due to explicit data mapping.

C. Performance of Execution Models

Our execution models focus on co-processor acceleration of larger-than-memory datasets, as described in Section IV. In this section, we evaluate the performance of these execution models with larger TPCH scale factors (with total input size

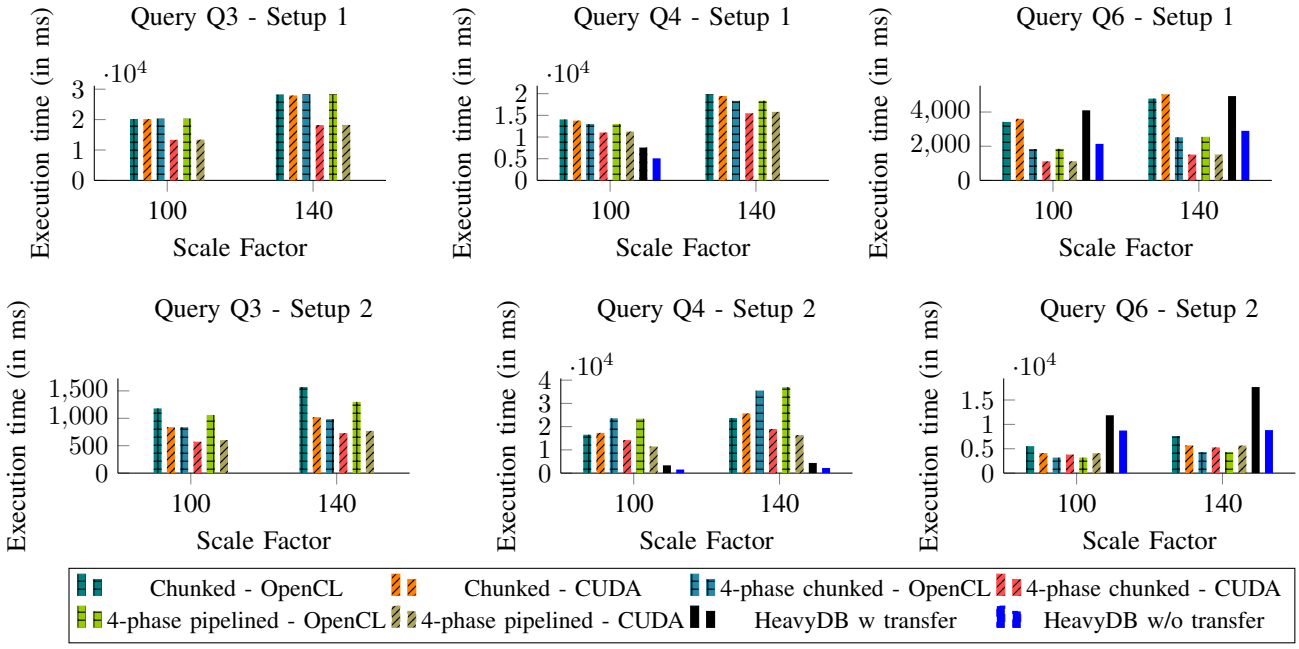

Figure 11: Performance of the execution models and HeavyDB execution with larger-scale factors

for queries varying from 2GB (2^{29} integer values) to 3.5GB ($2^{29.7}$ 32-bit integer values). Since multiple TPCH queries have similar patterns, we consider queries Q3 (multiple joins), Q4 (subquery), and Q6 (heavy aggregation) for our evaluation. We consider the size of chunks to be 2^{25} ints across all the queries. This chunk size is found to be optimal for the underlying GPU based on the available space in the device.

Overall, the plots in Figure 11 show the overall execution time for the execution model, with the underlying SDK and even the query being executed. In general, results show that 4-phase execution is faster than naive chunked execution in both OpenCL and CUDA. Mixing pinned memory with normal transfer consistently leads to better performance than the alternatives. Additionally, the results show that 4-phase pipelining produce similar improvements compared to four-phase chunked execution. This shows that the execution time of a query is so tiny that hiding it with transfer only provides minimal benefit. The rationale for the performance difference is as follows.

In particular, the query being executed on the target has the most significant impact on performance. For example, Q4 has an adverse impact on performance with 4-phase execution in OpenCL. This behavior shows that using pinned memory for multiple data transfers (which is the case with Q4) to a GPU degrades its performance. Since the query starts with building a hash table, there is no other operation between data transfer and hash build that has a considerable execution time for the transfer-time hiding. Therefore, the overall time for the pipeline is nearly the same as the data transfer time, leading to no performance benefit from attempting to hide execution time. Due to such poor execution behavior, 4-phased execution for OpenCL is nearly 2x slower than chunked execution. However, CUDA can overcome this issue and has up to 1.5x speed-up compared to chunked execution. This shows that the overhead of handling query execution using OpenCL incurs additional effort, which degrades overall performance. For Q3 and Q6, 4-phased execution is clearly faster than chunked execution. We see that the execution is nearly 2x faster for CUDA and 1.5x for OpenCL. Since these queries have a pipeline comparatively deeper than Q4, the 4-phase execution is beneficial. Therefore, depending on the query and its operators, the pipelined execution is beneficial. For the most part, four-phased execution has a speed-up of 3x (in the best case - Q6) until 1.3x (the worst case - Q3) over chunked execution. This performance difference is subject to change with newer GPUs. Next, we see that OpenCL performs worse in general compared to CUDA. As seen in Figure 9 and Figure 10, the difference in the execution of individual primitives, as well as the overhead of handling execution for OpenCL, leads to a higher execution time.

Comparison with HeavyDB: We compare our execution models with HeavyDB[‖] (formerly MapD [46]) both with a cold start (HeavyDB w transfer - with data transfer) and pure execution (HeavyDB w/o transfer). We use larger scale factors SF 100, 120, and 140 datasets for our evaluation. Since HeavyDB works with in-place tables in the GPU, Q3 cannot be executed for the given scale factors, as the hash table size exceeds the maximum capacity. For the other queries, we see that the in-place execution of HeavyDB is comparable with our chunked execution, whereas cold start is relatively slower than our execution models. In the case of Q4 and Q6, our

[‖]https://github.com/heavyai/heavydb

execution models show a performance improvement of up to 2x for in-place and up to 4x for cold start execution. This behavior can be associated with the delay in transferring a complete table to the device memory, whereas we only transfer chunks of the column necessary for execution. The transfer delay within HeavyDB can also be inferred from the difference in performance between cold and hot start.

In summary, using pinned memory for costly transfer and device memory for storing intermediate results benefits query execution. Furthermore, the results show a performance improvement of up to 3x the chunked execution. However, the benefits of such execution still depend on the query (and its pipelines). Furthermore, the execution of pipelining with transfer has a small impact since the transfer time dominates the execution of the overall query. Therefore, the hiding of execution time only improves a little. Finally, our results show that there is indeed a performance difference between OpenCL and CUDA, that hardware-sensitive implementation is vital for better performance.

Evaluation Summary: Overall, with our architecture, multiple SDKs can be plugged in, which allows us to have a common performance comparison module for database operations (Figure 9). Furthermore, we measure the overhead of handling these SDKs (Figure 10). Finally, our results from Figure 11 show that 4-phase chunked using CUDA and pinned memory is superior among our implemented execution models.

VI. Related Work

In this section, we compare our approach with other existing techniques for co-processor acceleration.

Query Engines for Co-Processors: Based on the underlying co-processor capability, many DBMS systems have been developed [12], [23]. Most prominently, various query engines are available for CPU-GPU coupled systems: Systems like GDB [30], Ocelot [31], CoGaDB [11], OmniDB [54], Saber [35], works in SQLite by Bakkum et al. [5] and many more (BlazingSQL, PG-Storm, etc.) [19] are examples of DBMS engines over GPU. Similar works on DBMS over FPGAs include work by Ziener et al. [55], DoppioDB [51], dbX [50], AxelDB [48]. Additionally, query engines exist over emerging co-processors like TPUs [32] and tensor cores [33]. These systems focus on optimal execution using the underlying co-processor, so they need extra portability support. Our system supports such extensions with its plugins. Components from all these systems, mainly operator implementations and device drivers, can be freely interchanged in our ADAMANT system and use scalable execution models. Further, one can combine operator implementation across these systems and link them to an optimal device driver for improved query execution.

Cross-Device Execution Models: Recent work like HAPE [18], HetExchange [17], Fluidic co-processing [27], work by Lutz et al. [37], Arefyeva et al. [2] explore an optimal query execution model of running GPU in tandem with CPU. Unlike these, our work focuses on execution models for processing using only co-processors.

Other than DBMS engines, abstract runtimes support general-purpose processing on a co-processor. Popular systems like StarPU [3], fluidic kernels [41], elastic computing [53], Kokkos [22], DAGuE [10], Parsec [9] support cross-device execution. However, these systems are not aware of the DBMS workload. Ours complement these with primitives and execution models to be DBMS conscious. Still, our ADAMANT can benefit from concepts in these systems to improve execution over a co-processor.

Task Model and Query Compilers: Finally, as mentioned previously, we rely on primitive definitions based on existing work. Some of such work that we closely relate to are Hawk [13], the work of He et al., Voodoo [43], and HorseQC [24]. These systems can be integrated into our ADAMANT system since proper primitive signatures are maintained.

Other than these, many works have a layered architecture for extensible query processing environments such as He..roDB [39], which also supports co-processor acceleration, Apache calcite [8], that extend pluggable interfaces for any data sources. Overall, ADAMANT varies from existing work in enabling a co-processor pluggable query engine with a pre-existing abstract query execution model.

VII. Conclusion

Hardware architectures are increasingly heterogeneous and many database engines are trying to utilize their capabilities for faster query execution. In this work, instead of developing a query engine from scratch over each of these devices or SDK choices, we propose a pluggable architecture that can plug in multiple devices and SDKs, with a low overhead. Our proposed architecture, for ADAMANT, has three layers that handle execution in a co-processor using granular database primitives. Along with our architecture, we also propose an execution model for scaling execution to larger than memory datasets. Based on our evaluation with CPU (OpenMP, OpenCL) and GPU (OpenCL, CUDA) prototypes, we observe that there is a marked overhead in handling OpenCL execution compared to OpenMP and CUDA. Furthermore, we also identify that our four-phased execution can be employed for significant performance improvements over chunked execution of up to 3x. Our experimental evaluation emphasizes the complex optimization space of queries in a co-processor environment, which (beyond the state of the art) spans further execution models, operator placement, and primitive implementation (including micro-optimizations or SDK choices), among more parameters. We hope that our ADAMANT prototype can facilitate the study of this optimization space for database systems. Overall, our ADAMANT query engine can be used to build an efficient query processing system around new hardware, with less effort.

Acknowledgment

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

References

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475. IEEE, 2006.

[2] I. Arefyeva, D. Broneske, G. Campero, M. Pinnecke, and G. Saake. Memory management strategies in cpu/gpu database systems: A survey. In *International Conference: Beyond Databases, Architectures and Structures*, pages 128–142. Springer, 2018.

[3] C. Augonnet, S. Thibault, et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2), 2011.

[4] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. *Proceedings of the Workshop on General–Purpose Computation on Graphics Processing Units*, pages 94–103, 2010.

[5] P. Bakkum and K. Skadron. Accelerating sql database operations on a GPU with cuda. In *GPGPU*, 2010.

[6] A. Becher, L. BG, D. Broneske, T. Drewes, B. Gurumurthy, K. Meyer-Wegener, T. Pionteck, G. Saake, J. Teich, and S. Wildermann. Integration of fpgas in database management systems: challenges and opportunities. *Datenbank-Spektrum*, 18(3):145–156, 2018.

[7] A. Becher, A. Herrmann, S. Wildermann, and J. Teich. Reprovide: Towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing. In H. Meyer, N. Ritter, A. Thor, D. Nicklas, A. Heuer, and M. Klettke, editors, *BTW 2019 – Workshopband*, pages 51–70. Gesellschaft für Informatik, Bonn, 2019.

[8] 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*, 2018.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[11] S. Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated dbms. *DBS*, 14(3), 2014.

[12] S. Breß, M. Heimel, et al. GPU-accelerated database systems: Survey and open challenges. In *TLDKS*, pages 1–35. Springer, 2014.

[13] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822, 2018.

[14] R. A. Bridges, N. Imam, and T. M. Mintz. Understanding GPU power: A survey of profiling, modeling, and simulation methods. *ACM CSUR*, 49(3), 2016.

[15] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward Hardware-Sensitive Database Operations. *International Conference on Extending Database Technology*, pages 1–6, 2014.

[16] D. Broneske, A. Drewes, B. Gurumurthy, I. Hajjar, T. Pionteck, and G. Saake. In-depth analysis of olap query performance on heterogeneous hardware. *Datenbank-Spektrum*, 21(2):133–143, 2021.

[17] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. Technical report, 2019.

[18] P. Chrysogelos, P. Sioulas, and A. Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *Proceesings of the 9th Biennial Conference on Innovative Data Systems Research*, number CONF, 2019.

[19] H. Chu, S. Kim, J.-Y. Lee, and Y.-K. Suh. Empirical evaluation across multiple gpu-accelerated dbmses. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–3, 2020.

[20] A. Drewes, J. M. Joseph, B. Gurumurthy, D. Broneske, G. Saake, and T. Pionteck. Optimising operator sets for analytical database processing on fpgas. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings*, page 30–44, Berlin, Heidelberg, 2020. Springer-Verlag.

[21] T. Drewes, J. M. Joseph, B. Gurumurthy, D. Broneske, G. Saake, and T. Pionteck. Efficient inter-kernel communication for opencl database operators on fpgas. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 266–269, 2018.

[22] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[23] J. Fang, Y. T. Mulder, et al. In-memory database acceleration on FPGAs: a survey. *VLDB*, 29(1), 2020.

[24] H. Funke, S. Breß, et al. Pipelined query processing in coprocessor environments. In *SIGMOD*, 2018.

[25] T. Gautier, J. V. Lima, et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *IPDPS*, 2013.

[26] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD international conference on Management of data*, page 215, 2004.

[27] T. Gubner, D. Tomé, H. Lang, and P. Boncz. Fluid co-processing: Gpu bloom-filters for cpu joins. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–10, 2019.

[28] B. Gurumurthy, D. Broneske, T. Drewes, T. Pionteck, and G. Saake. Cooking dbms operations using granular primitives. *Datenbank-Spektrum*, 18(3):183–193, 2018.

[29] B. Gurumurthy, D. Broneske, M. Pinnecke, G. Campero, and G. Saake. Simd vectorized hashing for grouped aggregation. In *European Conference on Advances in Databases and Information Systems*, pages 113–126. Springer, 2018.

[30] B. He, M. Lu, et al. Relational query coprocessing on graphics processors. volume 34. ACM New York, NY, USA, 2009.

[31] M. Heimel, M. Saecker, et al. Hardware-oblivious parallelism for in-memory column-stores. *VLDB*, 6(9), 2013.

[32] P. Holanda and H. Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–3, 2019.

[33] Y.-C. Hu, Y. Li, and H.-W. Tseng. Tcudb: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1360–1374, New York, NY, USA, 2022. Association for Computing Machinery.

[34] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint*, 2010.

[35] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, pages 555–569, 2016.

[36] I. Kuon, R. Tessier, and J. Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.

[37] C. Lutz, S. Breß, et al. Pump up the volume: Processing large data on GPUs with fast interconnects. In *SIGMOD*, 2020.

[38] S. Memeti, L. Li, et al. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *ARMS-CC*, 2017.

[39] M. Müller, T. Leich, T. Pionteck, G. Saake, J. Teubner, and O. Spinczyk. He..ro db: A concept for parallel data processing on heterogeneous hardware. In A. Brinkmann, W. Karl, S. Lankes, S. Tomforde, T. Pionteck, and C. Trinitis, editors, *Architecture of Computing Systems – ARCS 2020*, Cham, 2020.

[40] J. D. Owens, D. Luebke, et al. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, 2007.

[41] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices. In *ACM CGO*, 2014.

[42] M. Pinnecke, D. Broneske, G. C. Durand, and G. Saake. Are databases fit for hybrid workloads on gpus? a storage engine's perspective. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1599–1606, 2017.

[43] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *VLDB*, 9(14), 2016.

[44] C. Pohl. Exploiting manycore architectures for parallel data stream processing. In *GvDB*, 2017.

[45] 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*, 2015.

[46] C. Root and T. Mostak. MapD: A GPU-powered big data analytics and visualization platform. In *ACM SIGGRAPH*. 2016.

[47] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl. The Operator Variant Selection Problem on Heterogeneous Hardware. In *Proceedings of*

the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, 2015.

[48] B. Salami, G. A. Malazgirt, O. Arcas-Abella, A. Yurdakul, and N. Sonmez. Axledb: A novel programmable query processing platform on fpga. *Microprocessors and Microsystems*, 51:142–164, 2017.

[49] L.-C. Schulz, D. Broneske, and G. Saake. An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. *Proc. VLDB Endow.*, 11(11):1550–1562, jul 2018.

[50] T. C. Scofield, J. A. Delmerico, V. Chaudhary, and G. Valente. Xtremedata dbx: an fpga-based data warehouse appliance. *Computing in Science & Engineering*, 12(4):66–73, 2010.

[51] D. Sidler, Z. István, et al. doppiodb: A hardware accelerated database. In *SIGMOD*, 2017.

[52] H. K. H. Subramanian, B. Gurumurthy, et al. Analysis of GPU-libraries for rapid prototyping database operations: A look into library support for database operations. In *ICDEW*, 2021.

[53] J. R. Wernsing and G. Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. *ACM SIGPLAN*, 45(4), 2010.

[54] S. Zhang, J. He, et al. Omnidb: Towards portable and efficient query processing on parallel CPU/GPU architectures. *VLDB*, 6(12), 2013.

[55] D. Ziener, F. Bauer, et al. FPGA-based dynamically reconfigurable SQL query processing. *ACM TRETS*, 9(4), 2016.