doppioDB: A Hardware Accelerated Database

David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso
Systems Group, Dept. of Computer Science
ETH Zürich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Relational databases provide a wealth of functionality to a wide range of applications. Yet, there are tasks for which they are less than optimal, for instance when processing becomes more complex (e.g., matching regular expressions) or the data is less structured (e.g., text or long strings). In this demonstration we show the benefit of using specialized hardware for such tasks, and highlight the importance of a flexible, reusable mechanism for extending database engines with hardware-based operators.

As an example, we extend MonetDB, a main-memory column store, with a Hardware User Defined Function (HUDF) to provide seamless acceleration of two string operators, `LIKE` and `REGEXP_LIKE`, and two analytics operators, `SKYLINE` and `SGD` (stochastic gradient descent).

We evaluate our idea on an emerging hybrid multicore architecture, the Intel's Xeon+FPGA platform, where the CPU and FPGA have cache-coherent access to the same memory and the hardware operators can directly access the tables. For integration we rely on Hardware UDFs (HUDF), as a unit of scheduling and management on the FPGA. In the demonstration we show the acceleration benefits of hardware operators, but also their flexibility in accommodating changing workloads.

1. INTRODUCTION AND MOTIVATION

Relational engines exhibit great performance for a wide range of tasks. There are, however, well known operations and data types that cause problems. One of these data types is character strings which are both unstructured and expensive to process for anything but the simplest forms of pattern matching.

Most databases implement the SQL `LIKE` operator which can match multiple substrings divided by a wildcard '%'. For more complex string matching, some engines provide a vendor-specific regular expression operator, e.g. `REGEXP_LIKE`. In contrast to the string matching with a `LIKE` operator, regular expression evaluation is significantly more compute-intensive, resulting easily in a performance difference of an order of magnitude between the two operators.

© 2017 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

With the increasing amount of user-generated data stored in relational databases. There is not only the need to analyze unstructured text data, but general analytical operations in the context of machine learning become gradually more important to extract useful information from the waste amount of data collected. Many of these analytical operations incur a significant compute complexity not suitable to database engines where multiple queries share the available resources.

One way to address this trend is to use accelerators such as Xeon-Phi, GPUs, or FPGAs. Such approaches will often promise orders of magnitude performance improvements, however in many cases the integration into a real system voids these improvements because 1) the data needs to be reformatted to the execution model of the accelerator (GPUs, SIMD on Xeon Phi) and 2) the data needs to be partitioned between the host and accelerator memory. Indeed the integration into the database and addressing challenges related to data consistency and management is still an open problem.

Hybrid multicore architectures, such as Intel's Xeon+FPGA platform [1] and IBM's CAPI for Power8 [4], try to address these limitations. In these architectures, the accelerator is treated as another processor in the system and has direct access to shared memory. This architecture has the potential of removing both the data-reformatting and -partition overhead. In our work we take advantage of this tight coupling and implement Hardware User Defined Functions (HUDFs) which can access data in the database without explicitly moving data to and from the accelerator. By implementing the UDF interface the HUDF becomes just another operator from the point of the database engine and hides all the complexity of interacting with the hardware accelerator.

In this work we demonstrate the integration of multiple FPGA-based hardware operators, regular expression [3], skyline [5], and stochastic gradient descent, into MonetDB as explained in [3]. The hardware operators are fully runtime parameterizable, i.e., the chip does not need to be reprogrammed for executing new queries using the same operator. Thanks to their integration into MonetDB as HUDFs, they can be used seamlessly in queries. As we will demonstrate, our FPGA-based operators achieve at least 2-3x speed up over software running on a 10-core CPU, reaching more than an order of magnitude improvement in many cases.

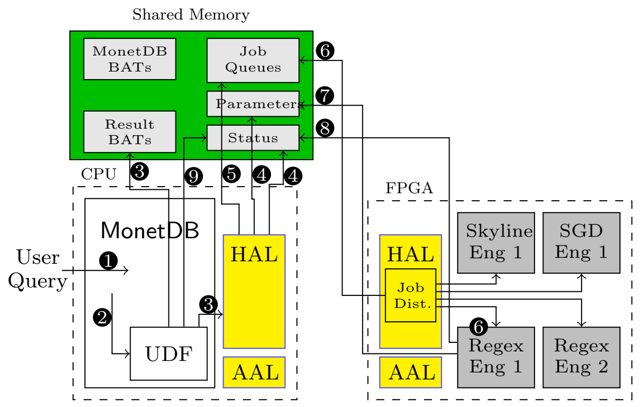


Figure 1: Overview of the system, the numbers show the steps of executing a regular expression query on the FPGA

2. SYSTEM OVERVIEW

The system used in the demonstration is described in [3]. As shown in Figure 1, it consists of three main parts: 1) MonetDB extended with our Hardware User Defined Function (HUDF), 2) the *Hardware Operator Abstraction Layer* (HAL) provides a simple software API to execute jobs on the FPGA and 3) four hardware engines which implement either the regular expression, skyline or stochastic gradient descent operator. Each hardware engine is runtime parameterizable such that it can adapt to the current query.

2.1 MonetDB

We integrate hardware operators into MonetDB using its native UDF interface. Unlike most databases which require the invocation of the UDF for each tuple, in MonetDB UDFs can operate on complete columns, called binary association tables (BATs). To guarantee that the operators on the FPGA can access the data in MonetDB, we altered MonetDB's memory allocation to use a custom memory allocator which manages the CPU-FPGA shared memory. In the current prototype system this region is limited to 4 GB, but this is not a fundamental limitation and should be lifted in future generations of the Xeon+FPGA system.

2.2 Hardware Operator Abstraction Layer

The HAL provides two main functionalities an API to construct and monitor jobs on the FPGA and the custom memory allocator for the CPU-FPGA shared memory region. After the initial handshake between software and hardware which is executed through Intel's AAL (Accelerator Abstraction Layer) library, all control communication is handled by the HAL. The HAL allocates all control data structures such as the job queue, job parameters, and job status in the shared memory region, thereby they are accessible from software and hardware. Similarly all the BATs, data columns, and intermediate results of MonetDB and the result BATs produced by the FPGA are allocated in this region.

When the HUDF in MonetDB creates a job through the HAL, a job is enqueued in the job queue in shared memory, where each operator type has its own queue. On the FPGA the *Job Distributor* is constantly monitoring these queues and assigning new jobs to available engines of this operator type. The HAL module on the FPGA also arbitrates the memory access from the four engines.

2.3 Execution Walkthrough

We want to illustrate the functionality of our system and the interaction of the three main components: MonetDB, HAL, and Hardware Engines through an execution walkthrough. The walkthrough explains the execution of a regular expression query, but the same steps apply to other hardware operators. The following steps are required when processing a user query, while the corresponding numbers in Figure 1 show where in the system they take place:

1. A query containing a regular expression is submitted.
2. As part of executing the query, MonetDB calls the UDF. The regular expression string and the input BAT are provided as parameters.
3. The UDF converts the regular expression into a configuration vector, allocates memory for the result BAT, and calls the HAL to create a new FPGA job.
4. The HAL allocates memory for the job parameters and job status data structures and populates them.
5. The HAL enqueues a job into the corresponding shared memory job queue.
6. The Job Distributor logic inside the HAL on the FPGA fetches the job from the job queue and assigns it to an idle Regex Engine (Engine 1 in this example).
7. The Regex Engine reads the parameters from shared memory and configures itself with the configuration vector. It then starts the execution and processes the input BAT.
8. After the engine terminates, it sets the `done` bit in its status memory and updates various statistics about the execution.
9. The UDF waits on the done bit and then hands the result BAT over to MonetDB.

Thanks to the standard UDF interface, HAL abstraction, and parameterizable regular expression operators on the FPGA, any regular expression given by a user query can be offloaded to hardware.

2.4 Regular Expression Engines

Each regular expression engine is capable of processing strings at 6.4 GB/s, with up to four engines leading to an aggregated bandwidth of 25.6 GB/s. However on the current platform the throughput is limited by the QPI link to around 6.5 GB/s, therefore deploying more than one shows only a slight improvement and deploying more than two shows no further improvement. The regular expression engines are parametrized before each query with a 512 bit configuration vector. This configuration vector is generated on the software side in the HUDF, more details can be found in [3]. Therefore the FPGA does not have to be reprogrammed to support multiple different queries.

2.5 Analytics Engines

The two analytics operators used for the demonstration are `SKYLINE` and `SGD` (stochastic gradient descent).

We integrated the skyline operator implementation of Woods et. al [5] into MonetDB as a HUDF. The skyline operator works on multiple columns and finds a list of records which are not worse than any other (i.e. they are part of the pareto optimal set). A common example, is a query over hotels which have a price and distance to the beach attribute. In this case the skyline operator would return all hotels which are not worse than any other hotel for these two attributes. Skyline is an iterative compute-bound operation with a variable runtime, similar to many machine learning algorithms. In our implementation the skyline operator can be parametrized at runtime to operate on up to 16 different attributes.

SGD is a very commonly used algorithm for training linear machine learning models. It is based on vector algebra, thus the inherent parallelism and deep-pipelined computation provided by an FPGA provides speedup over state-of-the-art CPU implementations. We integrate an SGD operator into MonetDB as a HUDF, so that linear model training can be performed on newly imported or already existing data in relational tables. The SGD operator supports data sets with up to 100,000 features and it is highly parameterizable (e.g., the convergence rate of the optimization, the frequency of model updates), so that the training can be tuned to the target data set to achieve an optimal convergence of the optimization problem.

3. DEMONSTRATION

3.1 Setup

For our demonstration we use version 1 of the experimental Xeon+FPGA system released under the *Intel-Altera Heterogeneous Architecture Research Platform*[1] program [1].

The system has two sockets and each socket is its own NUMA region. One of them contains a 10-core CPU (Intel Xeon E5-2680 v2) and the other an FPGA (Altera Stratix V 5SGXEA). In this experimental system it is only possible to install memory in the CPU's NUMA region which is equipped with 96 GB of main memory. The FPGA has cache-coherent access to the memory through the QPI bus. This memory access is clearly bound by the available QPI bandwidth which we measured to be around 6.5 GB/s for read-intensive workloads. The reason for this low bandwidth is partially in the prototype QPI endpoint which is implemented in FPGA logic and only runs at a frequency of 200 MHz. The QPI endpoint is part of the prototype system and cannot be modified. Based on announcements form Intel [2], we expect the memory bandwidth to increase significantly in the next generation of the platform.

The system runs Ubuntu 14.04 and a modified version of MonetDB (11.21.19) that includes all adaptations required to integrate the HUDFs.

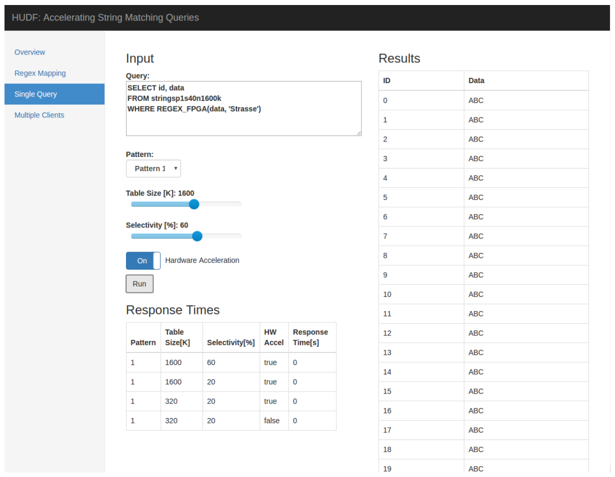
3.2 Scope and Presentation

For the demonstration the user can interact through a web interface with the database. The interface consists of two parts: 1) A simple form to submit single regular expression queries and retrieve the corresponding result and 2) a dashboard to deploy two different types of workloads and monitor the effect of hardware acceleration on the system.

A) Single Query

The web interface for this part is shown in Figure 2, the visitor of the demonstration will be able to choose among a varying number of queries and database tables. These vary

[1]Results in this publication were generated using pre-production hardware and software donated to us by Intel, and may not reflect the performance of production or future systems.



Query:
```
SELECT id, data
FROM stringsp1s40n1600k
WHERE REGEX_FPGA(data, 'Strasse')
```

Pattern: Pattern 1

Table Size [K]: 1600

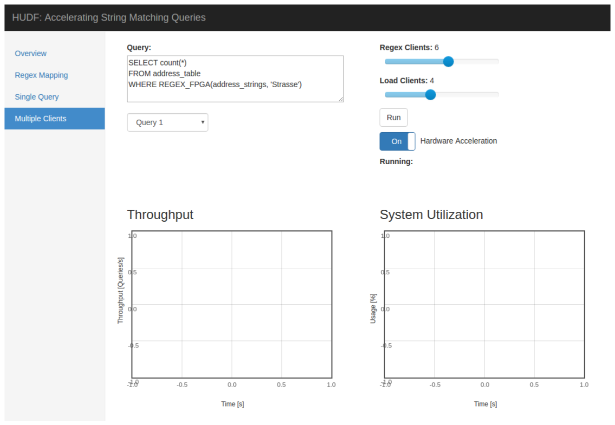
Selectivity [%]: 60

On Hardware Acceleration

Pattern	Table Size[K]	Selectivity[%]	HW Accel	Response Time[s]
1	1600	60	true	0
1	1600	20	true	0
1	320	20	true	0
1	320	20	false	0

ID	Data
0	ABC
1	ABC
2	ABC
3	ABC
4	ABC
5	ABC
6	ABC
7	ABC
8	ABC
9	ABC
10	ABC
11	ABC
12	ABC
13	ABC
14	ABC
15	ABC
16	ABC
17	ABC
18	ABC
19	ABC

Figure 2: Single Query dashboard



Query:
```
SELECT count(*)
FROM address_table
WHERE REGEX_FPGA(address_strings, 'Strasse')
```

Query 1

Regex Clients: 6

Load Clients: 4

On Hardware Acceleration

Figure 3: Workload acceleration dashboard

in regards to pattern complexity, size of the table, selectivity of the regular expression predicate and can be executed either with hardware acceleration enabled, software-only, or in a hybrid fashion between HW and SW. The visitor will see the different type of queries our system can handle and observe the effect of hardware acceleration through the reported response time. Additionally the results of the query are presented in the UI to the visitor, so they can verify their correctness.

In the case of the regex operator we will illustrate that the operator can be used even if the patterns in the selection is too large to fit on the deployed regular expression circuit on the FPGA. To still benefit from hardware acceleration it can be execute in a hybrid mode where it is partially evaluated on the FPGA and partially in software. As can be observed in the demonstration even a partial evaluation on the FPGA gives a significant performance boost over software-only evaluation.

B) Workload Acceleration

The interface used for this part of the demonstration can be seen in Figure 3. The user can deploy two different type of clients. The first type is executing a query using any of the hardware operators as chosen in the drop-down selection on the left. The second type executes simple queries to generate load on the system. The user has the option to either start only the clients running hardware operator queries or both. When the demonstration is running the web interface will fetch in intervals of 5 seconds the aggregated throughput of the clients as well as the current CPU utilization. To see the impact of hardware acceleration, the visitor can enable and disable it while the clients are executing the queries. The impact can then be observed in real-time through the changes observed in the graphs.

4. INSIGHTS FOR THE DEMO VISITORS

The demonstration should convey the insights we gathered related to the benefits and drawbacks of using a UDFs to interface with the accelerator. The abstraction of Hardware User Defined Functions (HUDFs) provide a seamless integration of hardware operators and hide the complexity of offloading to an accelerator from the database engine. This makes it possible to use the accelerator in many scenarios, and even compose its results easily with software operators (i.e. in case of hybrid execution of regular expressions).

However the UDF interface also imposes some limitations, e.g., depending on the database only one tuple at a time can be passed to the UDF, or the number of tables or columns a UDF can operate on is usually limited. An other important drawback, especially since we use UDFs to hide an accelerator, is that from the point of the database engine the UDF acts like a black box. Thereby making any predictions about its execution cost or runtime is nearly impossible in traditional database engines.

However, information regarding the accelerator such as capacity, current load, and a performance model are all available and could be made available to the database engine. This would mean exposing the accelerator as a more transparent unit and would allow the query optimizer to build a cost model. This is also an answer to the general problem of using accelerators when they indeed make execution faster. If, for instance, an accelerator is fully utilized but there are free cores on the CPU, it might make sense to execute the operator in software. To achieve a better integration with the query engine, the HUDF interface has to be extended further and the execution model of the accelerator has to be communicated in a way that is compatible with software. We plan to address these challenges in future work.

As for the choice of platform, in our work we used an Intel Xeon+FPGA system, one of the first high-performance shared-memory hybrid architectures. Given the announcements of future Xeon+FPGA systems [2] or the development of cache-coherent interfaces for accelerators such as OpenCAPI and CCIX, we expect to see more hybrid systems and an even tighter integration between accelerators and CPUs. As we showed in this work databases can benefit significantly from hybrid shared memory architectures, especially in regards to compute-intensive operations, and this benefit will only increase with tighter integration.

Acknowledgments

We would like to thank Intel for their generous donation of the HARP machine. Part of this work was funded through the Microsoft Joint Research Center MSR-ETHZ-EPFL.

5. REFERENCES

[1] N. Oliver, R. Sharma, S. Chang, et al. A reconfigurable computing system based on a cache-coherent fabric. In *ReConFig'11*.

[2] P.K. Gupta. Accelerating datacenter workloads. http://www.fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf.

[3] D. Sidler, Z. István, O. Muhsen, and G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD'17*.

[4] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. CAPI: A coherent accelerator processor interface. *IBM J. Research and Development*, 59(1), Jan 2015.

[5] L. Woods, G. Alonso, and J. Teubner. Parallel computation of skyline queries. In *FCCM'13*.