Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures

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

ABSTRACT

Taking advantage of recently released hybrid multicore architectures, such as the Intel Xeon+FPGA machine, where the FPGA has coherent access to the main memory through the QPI bus, we explore the benefits of specializing operators to hardware. We focus on two commonly used SQL operators for strings: LIKE, and REGEXP_LIKE, and provide a novel and efficient implementation of these operators in reconfigurable hardware. We integrate the hardware accelerator into MonetDB, a main-memory column store, and demonstrate a significant improvement in response time and throughput. Our Hardware User Defined Function (HUDF) can speed up complex pattern matching by an order of magnitude in comparison to the database running on a 10-core CPU. The insights gained from integrating hardware based string operators into MonetDB should also be useful for future designs combining hardware specialization and databases.

1. INTRODUCTION

Existing relational engines provide a number of basic mechanisms for processing strings with different trade-offs between generality and performance: LIKE is used to find (multiple) substrings, REGEXP_LIKE is used to evaluate regular expressions on strings and is more costly. Queries with CONTAINS run on a pre-built inverted index, which is quick but has additional costs in maintaining the index. Indexes on text (e.g., suffix trees, n-grams, or inverted indexes) occupy a substantial amount of space, require repeated lookups to match several patterns, must be updated on insertion, and in the case of the inverted index, have potentially stale data and need to be rebuilt periodically.

Using a commercial row store (referred to as DBx) and an open source column store (MonetDB) the above mentioned trade-offs between generality and performance in string matching can be precisely characterized. In Table 1, we show the response time when executing queries using CONTAINS, LIKE and REGEXP_LIKE. These clauses plug into the following simple query:

```
SELECT count(*) FROM address_table WHERE addr ...
```

As Table 1 shows, the response times increase by one order of magnitude steps with operator complexity (transitioning from an index lookup in CONTAINS to full regular expression matching with REGEXP_LIKE). Note that the performance of DBx and MonetDB should not be directly compared because DBx runs the query in a single thread, while MonetDB uses 10 threads. It is the common trend that we want to highlight.

Table 1: Comparison of string matching using different SQL commands in MonetDB and a commercial database, on 2.5 Mio. records

Query (WHERE clause)	Response time (s)	
Database	MonetDB	DBx
`CONTAINS('Alan & Turing & Cheshire')`	-	0.033
`LIKE '%Alan%Turing%Cheshire%'`	0.021	0.431
`REGEXP_LIKE('Alan.*Turing.*Cheshire')`	0.361	8.864

The problem of costly regular expression matching is well known but it is becoming pressing to solve given the increasing amount of text from social media enriching relational data. There is a trend toward using accelerators (i.e., be it many cores like Xeon-Phi, GPUs, or FPGAs), but most of these solutions have two shortcomings: 1) the data needs to be reformatted to reap the benefits of SIMD execution (e.g., Xeon-Phi) and 2) data needs to be partitioned between main memory and accelerator memory (e.g., GPUs or FPGAs on the PCI bus). While performance gains have been shown, the integration within a database engine is not straightforward because of these data locality, data formats, and data movement overheads.

Hybrid multicore architectures where reconfigurable hardware is embedded as a normal processor in the system reduce the above mentioned limitations. The Intel Xeon+FPGA platform [25] and IBM's CAPI for Power8 [35] are two examples of such hybrid architectures. The Intel Xeon+FPGA prototype system provides cache-coherent memory access over a QPI connection, CAPI does so over PCIe for reconfigurable hardware. The former is better suited for random access at cache-line granularity, the latter for coarse grained access. The shared memory architecture of these systems enables tight coupling between Hardware User Defined Functions (HUDFs) and the database engine without having to explicitly move data to and from the accelerator.

The paper makes the following contributions:

- Novel fully runtime parameterizable regular expression engine tailored to SQL
- Full integration of a hardware-based regular expression matching engine as an UDF into MonetDB.
- Showing predictable, complexity-independent performance for complex patterns with one to two orders of magnitude speed up over a 10-core CPU.

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035954

- Insights in how emerging hybrid architectures, e.g., Intel's Xeon+FPGA and IBM's CAPI, can be used to accelerate database operators and what their current limitations are.

In this work we design an accelerator with multiple parallel engines, and even though in our case they are all used for the same type of operator, in future systems one could partition the FPGA between different accelerator types. We believe these ideas are timely, given the trend in hardware specialization, e.g., the SPARC M7 which includes the on-chip data analytics accelerators (DAX), ASIC based automaton processors [7, 24], and Microsoft's Catapult [29] in data center scale deployments. The paper provides valuable data points for the development of future accelerators or processor extensions providing pattern matching functionality, similar to vector instruction units or specialized hardware engines. It also acts as an evaluation of the Xeon+FPGA system's usefulness for RDBMS workloads and provides some suggestions for its future versions. We found for instance the NUMA bandwidth to be too low, becoming a limiting factor of performance. This issue could be alleviated through the addition of cache-coherent memory on the socket of the FPGA.

2. BACKGROUND

2.1 FPGAs

Field programmable gate arrays (FPGAs) are hardware chips that can be reprogrammed arbitrarily many times and, once programmed, behave similarly to application-specific integrated circuits (ASICs) [36]. They are traditionally programmed using hardware description languages such as Verilog or VHDL, but recently high level languages and synthesis tools have emerged which enable translation of C/C++ or OpenCL code to logic gates [1, 42].

Since the on-chip static random access memory on FPGAs (called block RAMs or BRAMs) is limited to a few megabytes, standalone boards are often used as a "bump-in-the-wire" accelerator for stream processing (e.g., [21, 22]) to avoid storing much data or computational state. However, modern FPGA boards usually have DDR memory in the range of a few gigabytes attached to the FPGA, though at a higher access latency than the on-chip memory. An alternative to these designs is the hybrid system we consider here.

2.2 Intel Xeon+FPGA Prototype System

The system used throughout this paper is an experimental system released under the *Intel-Altera Heterogeneous Architecture Research Platform*[1] program [25].

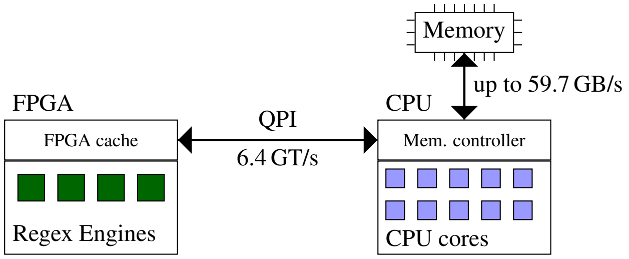


Figure 1: Architecture of the Xeon+FPGA prototype

As shown in Figure 1, this system has two sockets, one is occupied by a 10-core CPU (Intel Xeon E5-2680 v2) and the other by the FPGA (Altera Stratix V 5SGXEA). Each socket is a NUMA region, however in this system it is only possible to install memory in the CPU's NUMA region. Our machine has 96 GB main memory. In contrast to most other systems with accelerators, the FPGA has direct, cache coherent access to the main memory through the QPI bus. The FPGA accesses the main memory at 512 bit (cache line) granularity; we measured the peak performance to be close to 6.5 GB/s for most read-intensive workloads. On the CPU we measured up to 25 GB/s read bandwidth, less than the theoretical maximum in Figure 1. The reason for the low read bandwidth on the FPGA is only partially caused by the QPI link. The prototype QPI endpoint implementation on the FPGA limits bandwidth as well, since it only operates at 200 MHz. However, since this is an encrypted module provided with the prototype we cannot modify or improve it.

The system runs Ubuntu 14.04 extended with a special kernel module for communication with the FPGA. To interact with the FPGA, Intel provides an *Accelerator Abstraction Layer* (AAL) as a library. In an application that uses the FPGA, first a handshake between the software and hardware verifies that the desired *Accelerator Functional Unit* (AFU, in essence the user logic) has been instantiated in hardware. Then a so called Device Status Memory (DSM) page is allocated to share control and status information between the software and hardware. The DSM and any application memory that needs to be shared with the FPGA is allocated through the AAL library. This allocates memory at the granularity of 2 MB and pins it to contiguous physical regions. Pinning is necessary to ensure that accesses from hardware to main memory would never result in page faults. By default the Intel libraries allow the user to allocate a total of 2 GBs of memory in this fashion. By changing the kernel module and increasing the size of the pagetable on the FPGA we raised this to 4 GB. This is however just a current limitation of the libraries accompanying the prototype platform and the provided hardware modules. We expect this limitation to disappear in a future versions, in which almost all the memory should be shareable with the FPGA, similarly to direct memory access (DMA) for PCIe-attached devices, such as network cards or GPUs.

The FPGA is a standard device and can be programmed through the vendor's HDL tool chain. Intel provides a pre-compiled component that implements the QPI interface, including a 64 KB direct-mapped cache. The above mentioned virtual pages are accessed by the hardware through a pagetable data structure that is populated by software during memory allocation but resides on the FPGA on-chip memory (BRAM). The cost of virtual-to-physical address translation is constant and negligible.

2.3 Background on MonetDB

2.3.1 Data Storage Model

MonetDB is an open source column store. It stores relational tables as a collection of Binary Association Tables (BATs). A BAT consists of two columns, one containing an ID and the other the value `(OID,value)`. This data layout is convenient for fixed size data types such as integers, floats, etc. For variable length data types such as strings, MonetDB combines heap memory with a BAT containing offsets into this heap, see Figure 2. To access a string with a specific `OID` the database will first read the offset at the corresponding index in the BAT and then read the string at this offset in the heap memory. The heap also contains some meta-data and padding between the strings, but the length of the strings is not stored, instead they are null terminated.

MonetDB's BAT data structures are a good match for the FPGA's dataflow type of execution because they can be read sequentially,

[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.

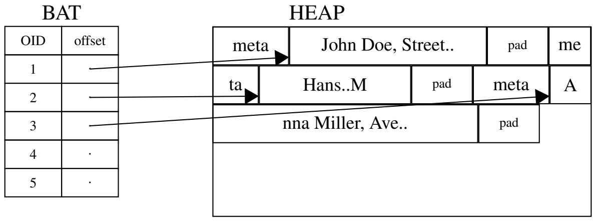


Figure 2: Data layout in MonetDB for variable-length data

which simplifies the FPGA logic. Note that in our case the logic accessing the strings is more complex as it needs to access a heap instead of a linear data structure.

2.3.2 *User Defined Functions*

User defined functions (UDFs) are a natural way for extending databases to implement a new functionality or to add an optimized version of certain database operators. UDFs present a promising way to extend off-the-shelf databases with hardware operators to accelerate complex queries. Currently UDFs in most databases have strict input and output interfaces, often working on scalar values instead of batches or BATs, and combining multiple columns of the same table or columns from different tables is not possible. In our work, this limits the type of operators which can be accelerated by the FPGA, ruling out for instance database joins, even though recent work shows promising steps in this direction [39, 11, 3]. We believe that having a more flexible UDF interface in databases, and also being able to provide a cost function for the UDF to the query optimizer could be beneficial for overall performance.

MonetDB provides the possibility to compile UDFs directly into the database engine. For our hardware based UDF, we use this interface as a wrapper to call the regular expression operators on the hardware. MonetDB's UDF interface further allows UDFs to operate on an entire BAT instead of single tuples. The UDF receives as an argument a pointer to a BAT, by operating on a complete BAT the overhead of calling the UDF is significantly decreased.

3. SYSTEM OVERVIEW

Figure 3 gives an overview of the complete system. On the CPU side, MonetDB is extended with a regular expression Hardware User Defined Function (HUDF). To provide a simple API for the MonetDB UDF to interact with the FPGA, we implemented a *Hardware Operator Abstraction Layer* (HAL) which uses Intel's AAL library. The FPGA contains the hardware part of the HAL and four Regex Engines. All engines operate concurrently and can process different queries. These engines have been designed to be parameterizable at runtime and require no reprogramming of the FPGA: the hardware implements a flexible circuit that is parametrized to the particular query at runtime. This is done using a configuration vector (512 bit memory words). In the current design, each engine is capable of processing data at the rate of 6.4 GB/s leading to a combined throughput of 25.6 GB/s. On the current experimental platform, the system throughput is limited by the QPI connection to around 6.5 GB/s. However this limitation should disappear in the next generation of the system [28].

Apart from the initial handshake between Software and FPGA, executed through Intel's AAL library, all control and data communication is done through shared memory managed by the HAL. We modified MonetDB such that database tables and all intermediate result BATs are stored in this memory. The same applies for result BATs produced by the hardware operators. Data structures managed by the HAL, such as the job queue, job parameters and job status, are also allocated in shared memory. On the hardware side HAL includes an arbitration module which guarantees fair access to the shared memory for each Regex Engine on the FPGA.

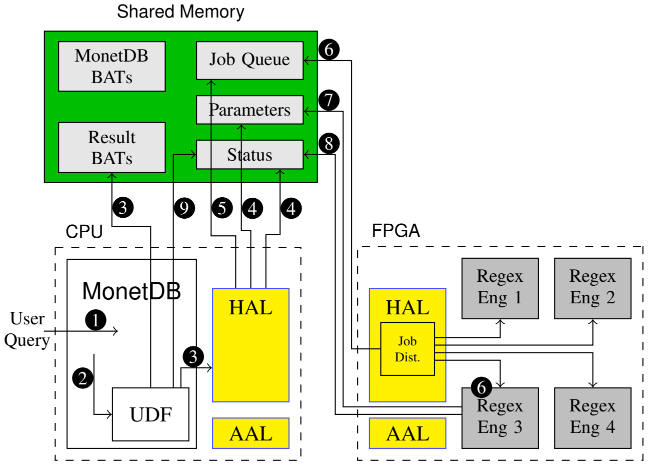


Figure 3: Overview of the system, the numbers show the steps of executing a regular expression query on the FPGA

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. The steps involved in offloading such a user query are explained below, while the corresponding numbers in Figure 3 show where in the system these steps 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 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 3 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.

Note that at no stage is the FPGA reconfigured. The design, as described, can run four concurrent HUDFs at a time, each of them for a different query. In the evaluation we provide a breakdown of the time it takes to allocate a HUDF, configure the Regular Expression Engines, and execute the complete operator.

In the following three sections we will explain in more detail, the UDF integration, the HAL abstraction, the Regex Engines and their internals.

4. INTEGRATION AND INTERFACE

To integrate the FPGA operator into MonetDB the following challenges had to be addressed: 1) Provide a seamless interface such that the hardware operator can be embedded into SQL, 2) Assure that the FPGA has direct access to the database memory without reformatting or partitioning data, 3) Provide an abstraction to MonetDB to easily execute and monitor operators on the FPGA.

4.1 Hardware User Defined Function

A common way of extending database functionality is through user defined functions (UDFs), and we chose this abstraction to ensure a seamless integration of the hardware operator into MonetDB. MonetDB does not support regular expression processing by default, but this functionality can be enabled before compile time. It uses the popular regex library *Perl Compatible Regular Expressions* (PCRE) to provide the REGEXP_LIKE operator in SQL. For simple string matching queries the more efficient LIKE operator can be used instead. Thanks to the UDF abstraction, our hardware operator takes the same arguments as the software based regular expression operator does and the two can be used interchangeably.

```
-- SW (substring matching)
  SELECT count(*) FROM address_table
  WHERE address_string LIKE '%Strasse%';

-- SW (regular expression)
  SELECT count(*) FROM address_table
  WHERE REGEXP_LIKE('Strasse', address_string);

-- HW (substring and regular expression matching)
  SELECT count(*) FROM address_table
  WHERE REGEXP_FPGA('Strasse', address_string)<>0;
```

The example queries above show how seamlessly the HUDF can be integrated into any SQL query (see REGEXP_FPGA above). The HUDF takes a regular expression pattern, e.g., "Strasse" or "(Josef|Klaus)strasse", and a column of strings or a string literal as parameters. The return type is short and in case the input was a complete column it will return a column of type short. A nonzero value means that a match was found in the particular string and the value represents the position of the match's last character. In case the value is zero no match has been found.

4.2 Hardware Operator Abstraction Layer

The Hardware Operator Abstraction Layer (HAL) is the abstraction between the HUDF in the database and the regular expression Engines on the FPGA. It is implemented on top of Intel's AAL library which is required to bootstrap the FPGA and to allocate the shared memory region. It provides two fundamental functionalities: *1)* a memory allocator to manage the 4 GB CPU-FPGA shared memory region, and *2)* an API to create, execute and monitor jobs on the FPGA.

4.2.1 Memory Allocation

Due to limitations of the hardware modules deployed in the prototype system and its companion kernel driver, the FPGA is not able to access the whole address space. Additionally the shared memory has to be pinned, since the FPGA is not able to trigger page faults. The FPGA has its own pagetable which is loaded once upon initialization. However, due to resource restrictions, this pagetable has a limited size. These restrictions limit how much memory can be allocated through the Intel AAL library to be shared with the FPGA. Currently this limit is 4 GB. To manage this memory region and make it accessible to MonetDB, we implement a custom memory allocator as part of the HAL library. This allocator uses slab allocation and manages internally multiple lists for different slab sizes. When MonetDB allocates memory through our custom memory allocator a slab that fits the requested size best is returned. By default MonetDB uses two mechanisms to allocate memory: for small objects (< 256 KB) it uses malloc and for larger objects, which are mostly BATs, it uses MMAP. We modified MonetDB to use our memory allocator to place every BAT in the CPU-FPGA shared memory, even if their size is smaller than 256 KB. This modification ensures that even BATs containing only a few tuples are allocated in the CPU-FPGA shared memory. Allocations smaller than 16 KB are however still handled through malloc as they represent metadata and other auxiliary structures which are not relevant to the FPGA operator.

4.2.2 Operator Execution on the FPGA

The UDF can execute and monitor jobs on the FPGA through the API of the HAL. For each job submitted through the API, HAL allocates a parameter structure and a status structure in memory (step 4, Figure 3). Apart from the configuration vector specifying the regular expression, the following parameters are written to the parameter structure: a pointer to the offset BAT, a pointer to the string heap, a pointer to the result BAT, the width of the offsets and the number of strings in the input. The memory pointers to these two structures are wrapped into a job descriptor and this is then enqueued in a shared memory queue which lies in the CPU-FPGA shared memory (step 5, Figure 3). On the hardware side HAL provides a Job Distributor module which observes the state of the Regex Engines and assigns new jobs from the job queue to the next available Regex Engine (step 6, Figure 3). The Regex Engine updates the status of the execution to the status structure in memory, this way it can be monitored by the UDF through the HAL API. Once the execution is finished the Regex Engine sets the done bit in this status memory, notifying the UDF that all results are written to the result BAT (step 8, Figure 3). Since currently the Intel AAL has no FPGA-to-CPU interrupt support, the UDF has to busy-wait on the done bit if low latency is desired. Once the UDF determined that the execution on the FPGA terminated, it can pass the result BAT to the database (step 9, Figure 3). MonetDB's execution model is based on BAT algebra operators which process a full BAT in a tight for-loop, as a result intermediate results between operators are fully materialized as BATs.

The pseudo-code below illustrates how the UDF calls the HAL API to start the operator. Before calling the HAL API, it allocates the result BAT and generates the configuration vector for the Regex Engine.

```
regexp_fpga(BAT **ret, BAT *src, char *regex) {
 // Convert regular expression into configuration vector
 unsigned char regex_config[CFG_VECTOR_WIDTH];
 fpga_regex_get_config(regex, regex_config);

 // Allocate result BAT
 BAT* result_ptr = BATnew(TYPE_void, TYPE_short, BATcount
      (src), TRANSIENT);

 // Create FPGA job through HAL
 FPGAjob job = create_regex_fpga_job(offset_ptr, heap_ptr
      , result_ptr, count, width, regex_config);

 // Wait for FPGA job to finish
 job.done();
}
```

A data arbiter in the HAL hardware module guarantees fair access to the shared memory for each Regex Engine by arbitrating the memory requests. Since each engine accesses memory in mostly sequential manner, the arbiter optimizes memory access by scheduling memory reads/writes from each operator in batches instead of single operations. The batch size of 16 is small enough to en-

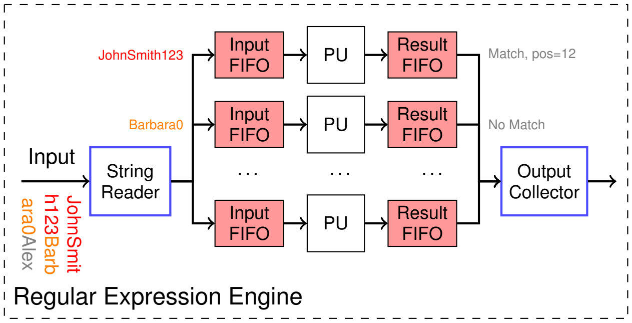


Figure 4: To reach high throughput, Processing Units (PUs) are grouped into an Engine. They operate on a scatter-gather fashion on the strings for the query.

sure good throughput without increasing memory access latency too much.

5. REGEX ENGINE ON THE FPGA

The execution of a HUDF corresponds to accessing one of the engines on the FPGA. Internally each engine consists of multiple Processing Units (PUs). Each PU is a non-deterministic finite state automaton (NFA) that matches a single regular expression to a series of bytes that constitute the input string. We use multiple PUs to parallelize on the input tuples. That is, each engine checks multiple tuples in parallel against the regular expression. The design of PUs builds on the state-of-the-art in FPGA regular expression matching [43, 37, 13] by adapting ideas to the needs of a database. Our design has three main properties that make it well suited for use in databases: *I) it is parameterizable at runtime* therefore it can evaluate any regular expression appearing in queries without reprogramming the FPGA, *II) it consumes the input at constant rate* regardless of pattern complexity or length which makes its cost function very simple, an important aspect for query planning, and *III) has an architecture optimized to text queries* that uses logic resources efficiently to match longer words inside regular expressions. As described in Section 8, there is a rich body of FPGA-based related work, but none of those fulfill all three goals above.

5.1 Assembling PUs into an Engine

In Figure 4 we show the internal structure of an engine. The first module in this pipeline is the String Reader which fetches the strings from memory. The String Reader receives two pointers as parameters, one pointing to a column of offsets, and the other pointing to the string heap (step 7, Figure 3). The offsets in the column point to the corresponding strings in the heap. During operation the String Reader alternates between two steps. In the first step it reads 512 cache lines (the depth of a BRAM FIFO on the FPGA) containing offsets from the offset column. Then in the second step it uses these offsets to fetch the strings from the string heap. The strings are parsed, aligned to the internal 512 bit data bus and forwarded in round-robin fashion to the cache-line wide Input FIFOs. At the end of the pipeline, the Output Collector collects the 16 bit match indexes from the Result FIFOs in round-robin fashion to guarantee that the results are in the same order as the input. Specifically, it collects 32 results and writes them into a cache line of 512 bit. These cache lines can then be written sequentially to the result column. The pointer to the result column is also provided as a runtime parameter.

The String Reader and Output Collector can operate at the rate of the QPI link, which in this system is bound at 6.4-6.5 GB/s. Since each PU is implementing an NFA, a single PU only consumes one input character (one byte) per cycle. Although it is possible to implement multi-character NFAs [43] which consume more than one character per cycle, their design is more complex and is not suitable for our flexible State Graph approach (explained in Section 6.2). In our default configuration, PUs are clocked at 400 MHz while the rest of the circuit runs at 200 MHz, therefore a single PU can process strings at 400 MB/s. Grouping 16 PUs together in a Regex Engine leads to a combined bandwidth of 6.4 GB/s which matches the bandwidth of the QPI and the String Reader. Using more PUs in an Engine would mean that they are starving on the input for more strings from the String Reader, while using less would mean that they could not keep up with the String Reader leading to backpressure and stalling in the pipeline. All PUs within an engine are connected to the same String Reader and Output collector, therefore they are all processing the same query and evaluating the same regular expression. PUs cannot dynamically be reassigned to a different String Reader or engine at runtime.

6. PROCESSING UNIT (PU)

Regular expressions are most efficiently expressed as state machines, in particular non-deterministic finite state automaton (NFAs) [9]. In contrast to deterministic finite state automatons (DFAs), NFAs allow multiple states to be active at the same time, matching well the parallelism of hardware circuits.

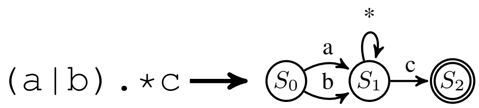


Figure 5: Translation from regular expression to NFA

Figure 5 shows a simple regular expression, `(a|b).*c`, translated to an NFA. Even this simple example shows non-determinism in that at the same time both state S_0 and S_1 might be active. In contrast, DFAs have additional states to avoid nondeterminism, which leads to the state explosion problem [41]. In software NFAs cannot be evaluated efficiently, since for each new input every active state has to be updated. Therefore the software implementation has to iterate over all states or keep track of all active states. On the other hand, hardware can efficiently implement an NFA since it can update all active states in parallel and take state transitions if necessary, all within a single clock cycle.

6.1 Parametrization

The main challenge in using FPGAs for regular expression matching in databases is that the cost of recompiling circuits and deploying them on the FPGA is prohibitively high. Therefore we designed the Processing Units (PUs) as generic NFAs which are parametrized at runtime to implement a specific regular expression. Figure 6 depicts the architecture of a single PU (for simplicity, it is scaled down to 4 characters and 4 states), it consists of Character Matchers and a generic State Graph. These are parametrized by the three registers: Tokens, Triggers, and State Transitions. The number of characters and states is fixed for a given deployment. This two parameters limit the space of regular expressions that can be mapped to a specific deployment, either by the length of an expression or its complexity.

Figure 6 also includes an example configuration vector encoding our example expression `(a|b).*c`. The configuration vector contains the characters, the bits representing the Triggers and State Transitions. Apart from that, it also contains flags which in-

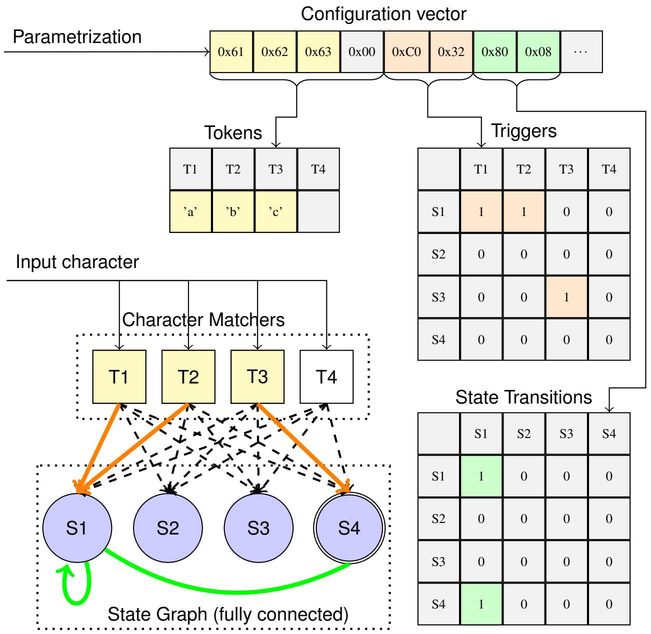


Figure 6: Internal structure of a PU with resources for matching 4 characters and four states. The configuration vector at the top parametrizes the registers (Tokens, Triggers, and State Transitions) on the right, implementing the regular expression `(a|b).*c`.

dicate if two Character Matchers are coupled together to evaluate a range instead of two separate characters. There are three parameter registers: Tokens, Triggers, and State Transitions. They are parametrized through the above mentioned configuration vector, and as a result the deployed PU implements the regular expression `(a|b).*c`. The Tokens parameter register defines the characters to be matched in the Character matchers, the Trigger defines which token triggers which state, and the State Transition register defines which state triggers which state. Being able to parametrize at runtime Tokens, Triggers, and State Transitions results in high flexibility when mapping regular expressions to the hardware.

The way we decouple the transition conditions from the NFA structure is inspired by previous work [37] on accelerating XPath queries for traversing XML document which already decoupled matching sequences of characters as logical *tokens*. Given the nature of XPath queries, the resulting NFAs have a much simpler one-dimensional structure (similar to a pipeline). As a result these NFAs cannot support certain regular expression operations, for instance alternations `(a|b)`, as used in Q_2 and Q_3 of our evaluation, are not supported. Furthermore it is not clear if their tokenizer can support the evaluation of ranges (`[0-9]`). In contrast to this fixed application-specific structure, our approach provides more flexibility through the flexible State Graph which can be modified at runtime.

6.2 Flexible State Graph

The main goal of our PU implementation is to deliver the same throughput as a hardcoded NFA while being able to evaluate any regular expression that fits into the deployed circuit (number of characters and states). To achieve these two seemingly conflicting goals, we deploy a structure similar to a fully connected directed graph on the FPGA, where each state is a node in the graph and the edges which represent state transitions can be enabled or disabled at runtime through the State Transitions register. The choice of which token triggers which state can be imagined as a bipartite graph, configured through the Triggers register. The circuit is implemented as synchronous logic. This means that all states will evaluate their inputs in parallel, make a decision, and then update their outputs all at once, based on a common clock. The same holds for the Character Matchers which update their outputs in sync with the same clock signal. Figure 6 shows how the State Transitions register is configured for our example expression `(a|b).*c`. The start states are implicitly defined by not having any activating edge, the end state is explicitly defined as the state with the highest index. Once it becomes active, a signal indicating a match is activated, this signal also contains the match index as a 16 bit unsigned integer.

6.3 Matching Sequences of Characters

To achieve flexibility we implemented the State Graph as a fully connected graph. This however puts some constraints on the hardware since each node needs to able to propagate their signal to all other nodes within a single clock cycle. Furthemore, the larger the State Graph grows the more space the circuit will take up on the chip, increasing quadratically. To reduce the required size of the State Graph, we match on sequences of characters in the regular expression instead of individual characters. This optimization is chosen because we expect natural language-related data and queries in the workloads. Extracting these sequences allows us to build a compacted NFA on so called *tokens* instead of individual characters. Inside a PU is a series of Character Matchers, as shown in Figure 6, that can be chained together to match a sequence of characters. We illustrate this optimization on the following expression: `(Blue|Gray).*skies` contains three tokens and the resulting expression would be: $(\alpha|\beta).*\gamma$. The trigger associated with token α is raised if the character sequence "Blue" has been detected. Assuming that the letter "B" was mapped to the first character box, then the trigger for α would be the output of the fourth character matcher (T4 in Figure 6). Apart from chaining Character Matchers together they can also be paired up to match a range defined by an upper and lower bound. The configuration of the Character Matchers is encoded with flags in the configuration vector.

6.4 Complex Expressions

In general, more complex regular expressions translate to NFAs with more states, and longer expressions require more character matchers. For instance, the expression `(Blue|Gray).*skies` translates to 3 NFA states and 11 characters. even though our regular expression matchers are runtime parameterizable, the number of supported states and characters is decided at deployment time (more details in Section 7.9). As a result, it is possible that a regular expression either requires more states or characters than available and therefore cannot entirely be mapped onto the engine. To resolve this issue and still benefit from hardware acceleration, we introduce hybrid execution, a technique which combines hardware and software execution to evaluate the regular expression. Hybrid execution is explained and evaluated in Section 7.8.

The regular expression matcher has been designed with a UTF-8 character set in mind and in our integration efforts we targeted the English character set. The circuits support case-insensitive collations or user-specified collations for matching strings that include characters with accents, etc. This is done by having multiple registers inside the character matchers that are compared in parallel to the input. These extra comparisons have no effect on performance due to the inherent parallelism of the circuit, but require additional resources even if not all queries use the collation functionality. The scalability of this circuit, and the cost of character matchers is discussed in detail in Section 7.9. Note that none of the existing work on regular expression matching on FPGAs considers the problem of collation.

7. EVALUATION

7.1 Experimental Setup

Our regular expression HUDF was deployed on the Intel system described in Section 2.2. Except for the PUs which are clocked at 400 MHz, our implementation runs at a frequency of 200 MHz, which is the clock frequency of the QPI interconnect logic on the FPGA. In our evaluation setup we deployed four Regex Engines on the FPGA, each provisioned to support a bandwidth of 6.4 GB/s. We use two different versions of MonetDB:*I)* an unmodified version 11.21.19, compiled with the default configuration for benchmarking, and *II)* a modified version that includes all adaptations required to integrate our UDF, built on top of 11.21.19.

BATs are allocated in the shared CPU-FPGA memory region. On the modified MonetDB version the `sql_optimizer` parameter is set to `sequential_pipe`, disabling intra-operator and dataflow parallelism, since they introduce an overhead when used together with our HUDF. The unmodified version uses the default parameter for the `sql_optimizer`, enabling multi-threading even for a single query. Apart from comparing to the unmodified version of MonetDB, we also compare to a commercial row store database (DBx). All experiments were executed on the Intel Xeon+FPGA prototype machine. Unless otherwise stated, each experiment was repeated 10 times and the reported number is the average of all runs. We found the standard deviation between repetitions to be very small, and consistent over all experiments.

7.1.1 Queries and Data

```
Q1:   SELECT count(*) FROM address_table
      WHERE address_string LIKE '%Strasse%';

Q2:   SELECT count(*) FROM address_table
      WHERE REGEXP_LIKE(address_string,
          '(Strasse|Str\.).*(8[0-9]{4})';

Q3:   SELECT count(*) FROM address_table
      WHERE REGEXP_LIKE(address_string,
          '[0-9]+(USD|EUR|GBP)');

Q4:   SELECT count(*) FROM address_table
      WHERE REGEXP_LIKE(address_string,
          '[A-Za-z]{3}\:[0-9]{4}');
```

Figure 7: Queries executed on address data

For our evaluation, we use four different string queries, see Figure 7, to capture the different complexity levels one might encounter in real workloads. Q_1 is a simple substring matching operation which can be expressed with the database operator `LIKE`, while $Q_2 - Q_4$ represent a regular expression matching operation which requires the database operator `REGEXP_LIKE`. For the FPGA implementation all queries are mapped to regular expressions and executed as such on the hardware.

Well established benchmarks, such as TPC-H and TPC-DS use only very simple string queries but they inspired our data set of shipment address strings. Each string contains a name, street, number, city, and area code. These parts are concatenated into a single string, as in the following example:

```
John|Smith|44 Koblenzer Strasse|60327|Frankfurt
```

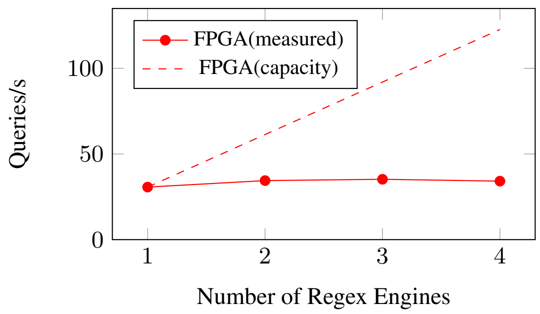


Figure 8: Throughput scaling with increasing number of Regex Engines

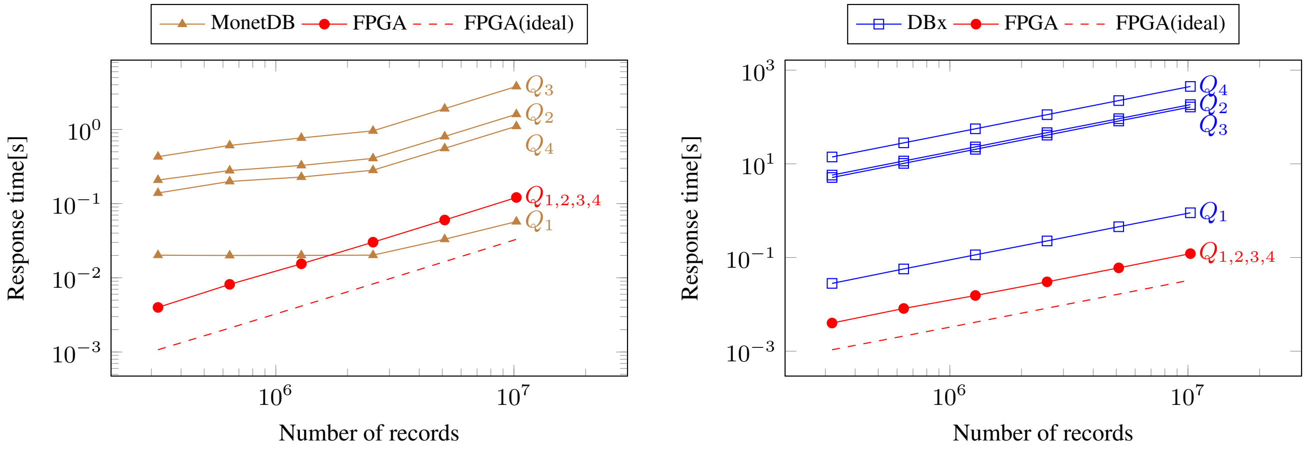
The strings are stored in a two column table where the first column is an `INT` ID and the second column of type `VARCHAR` stores the address string. If not otherwise stated, the length of the strings in our evaluation is 64 B. To have a well defined selectivity when running those queries, hits are inserted uniformly at random into the string data according to a predefined probability. The default selectivity in the following experiments is set to 0.2 unless otherwise stated.

7.2 Microbenchmark: Database operators

Table 1, presented in Section 1, shows the response times of executing a multi-substring pattern, using three different operators. As shown, the performance degrades quickly from CONTAINS to REGEXP_LIKE. The CONTAINS operator uses an inverted index which has to be pre-built ahead of query time and kept up-to-date by rebuilding it periodically (which takes more than 20 minutes for 2.5 Mio. tuples in DBx). Indexes can always be used to sped up a query if the queries are known beforehand. However with our work we want to support ad-hoc queries, which cannot make use of an index, therefore we assume that LIKE and REGEXP_LIKE operators work on un-indexed text data by scanning through the table. REGEXP_LIKE could be used for all four queries. However LIKE, which can be used for substring queries, performs significantly better. Therefore for Q_1 we use the LIKE operator, and for the remaining queries REGEXP_LIKE.

7.3 Microbenchmark: Regex Engine scaling

As discussed in section 5, a Regex Engine containing 16 Processing Units can process strings at a rate of 6.4 GB/s. However, the FPGA has to access the main memory which is located on the CPU socket over QPI. As a result the QPI bandwidth becomes a limiting factor, in a read-heavy microbenchmark we measured a read bandwidth of up to 6.5 GB/s over the QPI link. This means that a single Regex Engine can almost saturate it. This experiment should verify this assumption and show that using multiple engines in parallel has no negative impact on the aggregated throughput. In this experiment we use 10 clients to generate load by running Q_1 on a table containing 2.5 Mio tuples and measured the aggregated throughput (Figure 8). We can see that a single engine achieves 30.7 Queries/s which translates to around 4.7 GB/s of useful throughput, not including the 32 bit offset per string, meta-data, and padding in the heap. Accounting for these leads to around 5.89 GB/s read bandwidth. By adding a second engine the throughput increases slightly to 34.4 Queries/s, this result indicates that some memory latency can be hidden by using more than one Regex Engine. Although the String Reader is carefully designed to issue memory requests every cycle, there is some memory latency when switching form reading



(a) Comparison to MonetDB

(b) Comparison to DBx

Figure 9: Response time depending on input size and complexity

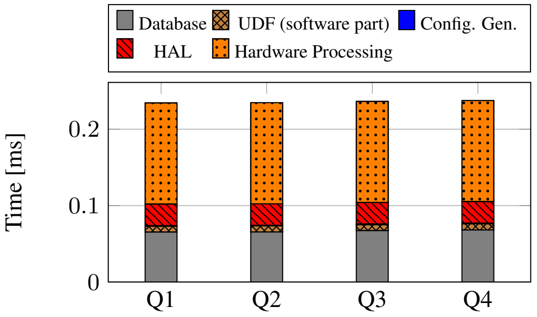


Figure 10: Breakdown of the response time, as spent in different parts of the system for a small relation with 10 k tuples.

offsets to reading the string heap (and back) which cannot be completely hidden. Adding more than two engines does not improve the aggregated throughput further, the system is already bound by the QPI bandwidth.

7.4 Microbenchmark: Response Time Breakdown

To determine the overhead of offloading a regular expression query to the FPGA we broke down the response time into different execution times occurring throughout the system. Figure 10 shows this breakdown for all four queries and illustrates the time spent in the database (everything but the UDF), in the UDF, to generate the Regex Engine configuration vector, create a job in the HAL, and finally the actual execution in the Regex Engine on the hardware. For this experiment we deliberately choose a table with only 10,000 entries such that the hardware execution time is not dominating the overall execution time. The time required to generate the configuration vector is less than 1 μs and therefore not visible in the plot. Before the Regex Engine is started, the HAL hardware module loads the job parameters from memory and parametrizes the Regex Engine including the PUs which takes around 300 ns.

7.5 Response Time depending on Complexity

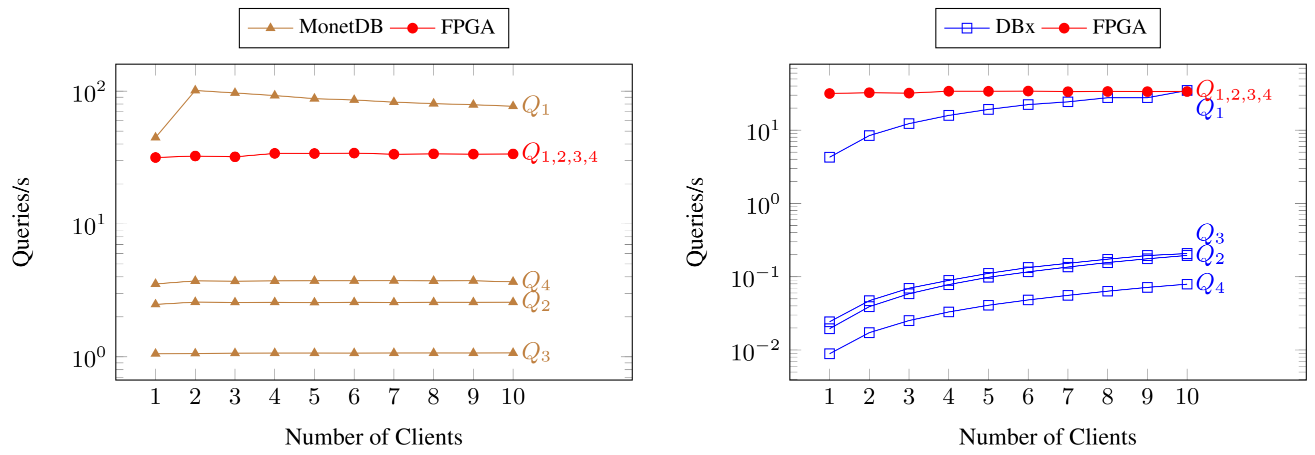
This experiment evaluates the impact of pattern complexity on the response time, while increasing the number of records in the input table from 320,000 to 10 Mio.. In Figure 9a, we compare the response time of MonetDB with the FPGA. Q_1 which has the lowest complexity and uses the LIKE operator performs well, slightly better than our UDF. However, the more complex queries Q_2-Q_4 have about an order of magnitude higher response time. On the FPGA, the performance is complexity independent and therefore the four lines plotted for Q_1-Q_4 are on top of each other. This graph also shows how MonetDB uses intra-operator parallelism by partitioning the data horizontally into 10 partitions, the number of CPU cores available. As a result, it achieves a constant response time up to 2.5 Mio. records, only for larger tables the response time increases linearly with the size. Although the FPGA parallelizes by horizontally partitioning the data to the four Regex Engines, it shows a linear behavior for any size of input. The parallelization and synchronization overhead for the FPGA is negligible, however it seems that for small input sizes the overhead of parallelization and synchronization in MonetDB leads to a constant response time. In Figure 9b the same comparison is done against DBx, the FPGA numbers are the same as in the previous graph. Similarly to MonetDB, the response time for the simple query Q_1 is more than an order of magnitude lower than for the complex queries. Unlike MonetDB, DBx uses strictly one thread per query which means this experiment is single-threaded. As a consequence the response time scales linearly with the input size.

In both of these experiments the hardware UDF is clearly bound by the QPI bandwidth, the possible response time of the FPGA without bandwidth limitation is indicated by the dashed line in the graphs. In which case our UDF would be competitive even for the low complexity query.

7.6 Throughput

In the previous experiment, only a single client was sending requests to the database. As a result DBx was not able to take advantage of all available CPU cores. In this experiment we keep the input size fixed to 2.5 Mio. records and increase the number of clients sending requests. We report the throughput and compare in Figure 11a with MonetDB and in Figure 11b with DBx. Again, we see that MonetDB performs better than the FPGA for the simple query Q_1. For the complex queries MonetDB performs about 5-15x slower than Q_1 or the FPGA. The FPGA can deliver constant throughput independent of the number of clients.

While MonetDB with its multiple levels of parallelism delivers almost constant throughput independent of the number of clients,



(a) Comparison to MonetDB

(b) Comparison to DBx

Figure 11: Throughput with increasing number of clients, number of records in table 2.5Mio.

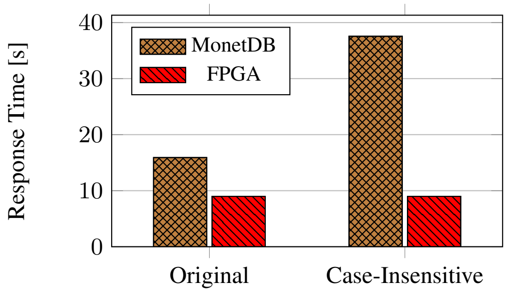


Figure 12: Response Time running TPC-H Query 13, comparing LIKE and ILIKE in MonetDB with the FPGA

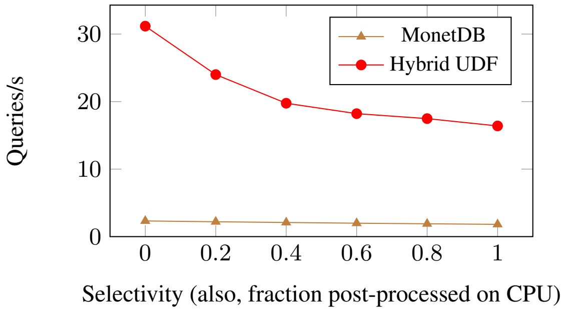


Figure 13: Hybrid execution of query Q_H

DBx which strictly assigns one thread per query shows a linear increase with the number of clients. For the simple query Q_1 it can match the throughput of the FPGA when serving 10 clients.

7.7 Complex Query

This experiment illustrates how our regular expression operator can be seamlessly integrated into any type of query over string data. Only a few queries in the TPC-H benchmark suite are using the LIKE clause. For this experiment we choose the TPC-H Query 13. Due to the limited memory space the scaling factor to generate the data was set to 0.1.

```
TPC–H Q13:
SELECT c_count, COUNT(*) AS custdist
FROM (
        SELECT c_custkey, count(o_orderkey)
        FROM customer
        LEFT OUTER JOIN orders ON
                c_custkey = o_custkey
                AND o_comment not like '%special%requests%'
        GROUP BY c_custkey
        ) AS c_orders (c_custkey, c_count)
GROUP BY c_count
ORDER BY custdist desc, c_count desc;
```

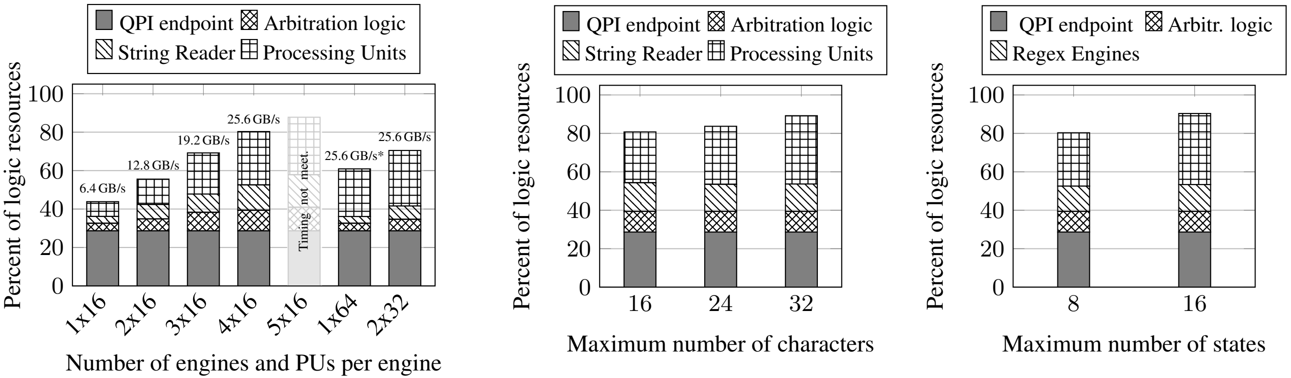
Apart from executing the query in the original version with the LIKE clause, we also modified it to use the ILIKE clause which enables case-insensitivity. As can be seen in Figure 12, using the case-insensitive version slows down the query execution in MonetDB by a factor of 2. On the other hand the regular expression operator on the FPGA is about 30% quicker than the LIKE, and it can provide case-insensitivity without any overhead.

7.8 Hybrid Execution

As explained in Section 6.4 there are two limitations on the regular expressions that can be mapped onto our regex engines: the maximum number of the characters in the expression and the number of states required to model the NFA. While the expressions we use in the evaluation fit on the device, for cases when they do not, the idea is to split regular expressions into two pieces at a suitable point, e.g., at the occurrence of a wildcard `'.*'`. If one of these pieces fits into the regular expression matcher on the FPGA, we can pre-process all tuples on the hardware, and post-process the ones that matched (the remainder of the expression) on the CPU. This idea resembles recent work on GPUs by Pirk et al. [27] in which the authors perform hybrid computation between GPU and CPU, by partitioning data between the two bitwise. That is, the GPU holds the most significant bits of the dataset and perform for instance a pre-selection, after which the CPU does a pass over the remaining data.

To determine the performance of this hybrid execution, we constructed a new query which extends the regular expression from Q_2 with an additional substring:

```
QH:
SELECT count(*) FROM address_tables
WHERE REGEXP_LIKE(address_string,
'(Strasse|Str\.).*(8[0-9]{4}).*delivery';
```



(a) Resource usage scaling with the number of Regex Engines and Processing Units

(b) Resource usage in a `4x16` setup with increasing number of characters for a 8 state deployment

(c) Resource usage in a `4x16` setup with increasing number of states for a 16 character deployment

Figure 14: Scaling of resource usage

This regular expression can easily be split into two parts at the wildcard. The first part is evaluated on the FPGA and the substring search for 'delivery' is executed in software. We created the dataset such that all strings matching the first part will contain the last part as well. This means that the selectivity of the pattern corresponds to the percentage of strings that would need post-processing in a hybrid setup. This query was executed on a table containing 2.5 Mio tuples. Figure 13 shows the throughput of this approach. The post-processing for the hybrid version occurs on the software-side of our regular expression UDF. In comparison to MonetDB, the hybrid execution achieves a speedup of up to 13x. This shows that even if a regular expression cannot be entirely evaluated by the FPGA, it might still be beneficial to run part of the expression on it. The deterministic performance of the hardware also helps in deciding when and how to use hybrid execution.

7.9 Resources

On FPGAs every functionality takes up real estate and therefore one of the key metrics is the total hardware resource usage of the implemented circuit. For simplicity, we will group resources into two categories: Logic and BRAM. The former stands for resources used to implement behavior while the latter is used for storage, configuration and FIFO buffers.

Although a single Regex Engine is designed to saturate the QPI bandwidth, it is possible to deploy up to four of them on the FPGA, as shown in Figure 14a. Thereby we make use of the available resources and enable concurrent execution of up to four queries on the FPGA. The Figure shows that even five engines can fit onto the FPGA, however the routing tools cannot find a valid routing which meets the timing requirements of the circuit.

Our default configuration assembles 16 PUs into a single Regex Engine leading to 6.4 GB/s bandwidth per Engine, matching the available memory bandwidth. However, there are two possible alternatives to our default configuration of `4x16`, with different trade-offs regarding concurrency, throughput, and resource usage: two engines with 32 PUs each and one engine with 64 PUs. In the latter one the PUs would starve on the input side, since the String Reader at best can produce one cache line per cycle at 200 MHz leading to 12.8 GB/s. For the former alternative the current String Reader would be sufficient, though the available memory bandwidth means that also in this configuration PUs are starving on the input side. Deploying less than 16 PUs would mean the String Reader produces more strings than can be processed leading to stalling in the processing pipeline.

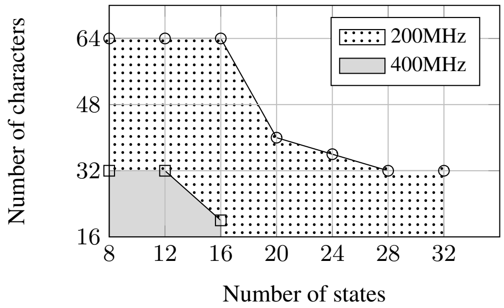


Figure 15: Trade-off between regular expression complexity and throughput. For the points outside the marked area the tools were reporting timing violations.

A large part, 28%, of the resources is consumed by the QPI endpoint which is implemented in FPGA logic. This overhead is constant for any of the above mentioned PU/Engine configurations. The control logic for data arbitration and the String Reader logic scales with the number of engines, while the logic consumed for the Regex Engine is related to the number of PUs it contains. Our default configuration can process data at a rate of 25.6 GB/s about four times the available bandwidth of QPI, while using 80% of the available logic resources. In the future, the following factors could increase the number of engines that can be deployed or, alternatively, increase the maximum clock frequency at which they can be operated: more resources on the FPGA, improved routing in next generation chips [12], or hardening of the QPI endpoint including its cache.

To determine the scalability of the PUs, we first evaluate the resource requirements when either increasing the number of characters or the number of states. In Figure 14b we show that, as expected, increasing the maximum number of characters has a linear effect on resource usage and that even with a high character count the circuit fits on the FPGA. Not shown in the figure are the BRAM consumption numbers, because they are constant at 42%. Also, it is clear that the QPI endpoint itself occupies a significant portion of the chip, 28% of logic and 4% of BRAM resources. The effect of increasing the state count is more accentuated (Figure 14c), because the logic resources grow quadratically with the size of the fully connected graph. With up to 16 states in the regular expression a significant portion of the chip is used by the State Graph.

In addition to the configurations discussed above, we explore how much additional complexity (states and characters) could be implemented on the FPGA if the PUs would be clocked at a lower frequency. Reducing the frequency reduces the throughput as well, but allows for longer signal delays between different parts in the PU. This means the State Graph can be larger and still meet the timing constraints. For this evaluation the `2x16` setup is used to make sure that enough resources on the chip are available to increase the complexity. Figure 15 shows the space (number of states and characters) of valid PU configurations for the two target frequencies. By halving the clock frequency, the space of possible configurations is significantly increased. Although the clock frequency is reduced, the circuit can still deliver enough bandwidth to saturate the current QPI bandwidth. Note that these results can change even for minor changes on the circuit design. Furthermore the tools are non-deterministic, this means there is a small variation between different synthesis runs. Nevertheless this evaluation illustrates the trade-off between clock frequency/throughput and complexity.

8. RELATED WORK

There are several types of string-based queries in databases: string matching, regular expression matching, approximate string matching, and document search. The work presented in this paper addresses the first two, focusing mostly on regular expressions. Due to their high computational overhead, regular expressions are not used very often in databases. Reducing their cost could enable more complex queries and varied workloads. Aside from relational databases, regular expression matching is used in many applications, such as network intrusion detection (NID) [20, 8], compilers, and DNA sequencing [19]. Standard libraries of most programming languages include regular expression matching functionality.

8.1 String Matching

String matching is widely adopted in databases through the standardized SQL LIKE clause. It can be used to match multiple substrings divided by the wildcard character '%'. Different type of indexes are used to improve string search, such as suffix, n-gram or inverted index. Generally this indexes have a large memory footprint exceeding the size of the indexed data. Oracle and DB2 support regular expressions through REGEXP_LIKE and full-text search through CONTAINS. Even though they both use a special index for the latter, the exact semantics of the operator are vendor specific. Microsoft SQL Server extends the LIKE operator with ranges, e.g., `[a-d]` can match `a, b, c` or `d`.

For software there are two efficient algorithms Knuth-Morris-Pratt (KMP) [17] and Boyer-Moore (BM) [2], the latter one performs generally better because it can skip over larger parts of the input string than KMP and therefore is often used as a benchmark in string matching literature. Although simple string matching can be executed efficiently on CPUs, as both DBx and MonetDB demonstrated in the Evaluation section, there is work showing how GPUs can be used to further accelerate this task. Sitaridi and Ross [34] use GPUs to accelerate substring search in strings, however their work assumes that the data is already in the GPU's memory and has a very specific layout which suits the GPU's execution pattern. When running on string data that has not been organized specially for the GPU the performance drops from 60-70 GB/s to around 20 GB/s, comparable to CPU-based execution. Additionally, the more wildcards the expression has, the slower the matching becomes. In contrast, our specialized hardware solution provides complexity-independent performance on unmodified data.

8.2 Regular Expressions

Evaluating regular expressions is costly in software, as the execution easily becomes compute bound. The regular expressions are most often mapped to deterministic finite automaton (DFA) because these only have one possible transition per parsed character, leading to more predictable performance. On the other hand, DFAs suffer from state explosion (illustrated for instance in [41]). Some databases provide native functions for regular expressions and in other systems this functionality can be added through User Defined Functions (UDF).

Hung et al. [18] use GPUs to accelerate regular expression matching. Their approach uses DFAs enhanced with techniques to reduce the effect of state explosion, and report on-device throughput numbers of 10-15 GB/s depending on the pattern complexity. Once the data movement over PCIe is also taken into account the throughput drops roughly by a factor of 10, putting it in the range of software-based regular expression implementation. Unlike with PCI-based accelerators, the communication overhead in our shared memory CPU-FPGA system is negligible.

In their recent work, Sitaridi et al. [33] show how careful optimization of DFAs to a SIMD execution model results in faster regular expression processing. Additionally, they propose an early-exit mechanism that helps in cases where the regular expression has to match the entire input string (e.g., e-mail address checking). However, the software trade-offs mentioned before still hold: both on regular CPUs and many-cores (such as the XeonPhi used in their paper) performance drops with increased query complexity (size of the DFA). Furthermore, if the execution cannot take advantage of the early exit technique, it must process all bytes of the input resulting in a throughput of around 6 GB/s on a 4 core CPU. In a workload most similar to our evaluation the Xeon Phi achieves around 30-40 GB/s of throughput, benefiting from the high memory bandwidth of the on-chip GDDR5 RAM. In contrast to our work, they do not discuss how the XeonPhi co-processor could be integrated into an application, such as a database engine, and whether this would impact performance.

There has also been progress in novel algorithms for parallel regular expression matching, described in [38]: vector instructions are used to pre-compute state changes, by enumerating possible state transitions. Although, this approach increases the overall computation per expression, this can be compensated by the use of multi-threading and SIMD parallelism leading overall to a speedup. While promising, this work shows most improvement when using relatively simple expressions, and reaches a throughput of 5-5.5 GB/s. In the database scenario generally multiple queries have to be served, therefore it is more beneficial to use the multi-threaded parallelism to execute multiple queries in parallel than speedup the regular expression evaluation of a single query by pre-computing a large number of potential transitions.

8.3 FPGA State-of-the-art

Regular expressions are very common in network intrusion detection systems. Those systems have to match each network packet against multiple rules (e.g. Snort [30], Bro [26]) to determine if the packet is malicious. Work in this area [4, 20, 32, 41] focuses on fusing many different regular expressions, all known beforehand, in a way that they can all be matched against the same input. In our work we focus on answering a single regular expression, not known beforehand, aiming at much higher data rates than what are common in networking related work (6-25 GB/s vs. 1-10 Gbps).

Many FPGA related work focuses on packing the regular expressions as tightly as possible, that is, to use as little resources as possible, e.g. [43], and also exploring ways of increasing performance by consuming more than one input character per clock cycle. The work of Teubner et al. [37] explores ways of compactly packing a matching circuit on the FPGA for answering XPath queries on XML data. To increase logic sharing they use a re-programmable tokenizer component. Due to the way XPath queries are structured, their work requires the ability to change the textual content of a query but not its structure. As a result the structure of the NFA in hardware is fixed in a pipeline shape. In contrast, we provide a flexible and compact representation for the NFA that enables more parallelism. Other works [15, 13] look at making the structure of the finite state automaton more flexible, so that it can be modified at runtime. While showing promising results, these efforts studied the problem without integration in a larger system. In this work we combined ideas from the state-of-the-art, while focusing on the needs of the database. We provide a simple API to access the regular expression engines on the FPGA and wrap them in UDFs, which provides the best of both worlds: the flexibility of software and state-of-the-art regular expression matching performance.

8.4 SQL Operators on FPGAs

There is an extensive body of work on implementing SQL operators in reconfigurable fabric, ranging from predicate evaluation [31, 23], through join [11] and aggregation with group-by [5] to histogram building [14].

In accelerator based approaches, the specialized hardware (GPU, many core devices, FPGAs, etc.) is either connected through the PCI bus (e.g., [16, 34, 29]) or placed on the data path, for instance the network card [10] or between disk and CPU [40, 6]). In the latter approach the accelerator is not incurring additional data movement, since the data has to be moved from the storage/network to the CPU either way. Ibex [40] an intelligent storage engine for SQL off-loading adopted this method to avoid data movement. Ibex can offload selection, projection and group-by aggregation to the FPGA which is placed at the storage device. One challenge of placing the FPGA in the data path is the limited amount of state that can be kept in the device. Therefore many approaches [41, 21, 22] focus on stream processing where only a limited window of the stream has to be considered. A runtime configurable approach was introduced by Najafi et al. [22] which use *Online Programmable Blocks*. These blocks can be configured for different SQL operations for each new query without reconfiguration of the FPGA.

9. DISCUSSION

In this work we showed how emerging hybrid multicore architectures can enable seamless integration of hardware accelerators into database engines. Having consistent data access from the accelerator is crucial in the database domain to avoid partitioning, reformatting, or additional movement of data. Although the experimental platform under evaluation has certain limitations such as limited memory bandwidth, limited memory capacity, and lack of low-latency interrupts/communication between FPGA and CPU, it still provides a good indication of the next generation systems and how they can be used to the benefit of existing database engines. In fact, Intel already released some details [28] about the next generation Xeon+FPGA architecture which will address the issues of memory bandwidth by providing both a QPI and PCIe link to the FPGA.

In future systems we also expect improved (virtual) memory management mechanisms and that the FPGA can access the full address space of the machine. Both of these functionalities are already available on IBM's CAPI platform where the FPGA can send pagefault interrupts to the operating system. Combined with recent industry initiatives such as OpenCAPI and CCIX we expect to see more architectures where accelerators have direct and cache-coherent access to shared memory.

For a seamless integration of our hardware operator we used the UDF interface common in many databases. However, it has a few limitations: 1) from the perspective of the database the UDF acts like a blackbox which limits the ability of the query optimizer to take the right decisions, 2) the query optimizer has no knowledge about the capacity or current load on the FPGA especially when many concurrent queries are executed, 3) the UDF cannot provide a cost model to the database. Based on this, it is nearly impossible for the query optimizer to predict the execution time of the FPGA-based UDF. Additionally, in most databases the UDF interface only allows processing of a single tuple at a time leading to a high invocation overhead and reduced performance. MonetDB as an exception provides a BAT-based interface. This drastically reduces the overhead of calling the UDF, which otherwise could void the benefit of specialized hardware.

Given the current trend towards customized hardware we expect that databases will have to adapt to take full advantage of such accelerators. Either by enhancing the UDF interface to address the aforementioned shortcomings or through new execution models for database engines which provide native support for accelerators. This way many compute-intensive operators could benefit from hardware acceleration. Combined with partial reconfiguration of the FPGA, the database engine could deploy multiple different hardware operators at runtime according to characteristics of the current workload. The query optimizer will then be able to dynamically decide where an operator with both a hardware and software implementation, such as our regular expression operator, will be executed. This decision can be taken depending on the operator's cost model and current load in the system.

10. CONCLUSION

In this paper we looked at regular expression matching, a compute-bound operation in database engines. We implemented an FPGA-based runtime parameterizable regular expression operator using state-of-the-art techniques. This operator was integrated into an existing column store, MonetDB, as a hardware user defined function (HUDF) on a hybrid CPU-FPGA multicore machine. With the introduction of more shared memory hybrid hardware platforms we see more opportunities to use specialized hardware, such as FPGAs, to accelerate compute-intense database operators and integrate them as UDFs into the database engine. Therefore we consider the lessons learned in this work applicable for other compute-intense operators.

Our regular expression HUDF not only shows significant acceleration of query execution both in comparison to MonetDB and a commercial row store, but also provides predictable performance independent of regular expression complexity. We further explored the trade-offs of our hardware circuit regarding resource usage, throughput, and supported regular expression complexity.

The source code of the hardware operator, the integration into MonetDB, and all related software is released as open source[2]to facilitate further exploration of these new architectures.

Acknowledgments

We would like to thank Intel for their generous donation of the HARP v1 prototype. Part of the work of Zsolt István has been funded by Microsoft Research.

[2] http://github.com/fpgasystems/doppiodb

11. REFERENCES

[1] Altera. Programming FPGAs with OpenCL. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[3] R. Chen and V. K. Prasanna. Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform. *FCCM'16*.

[4] C. Clark and D. Schimmel. Scalable pattern matching for high speed networks. In *FCCM'04*.

[5] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *FCCM'13*.

[6] J. Do, Y.-S. Kee, J. M. Patel, et al. Query processing on Smart SSDs: Opportunities and challenges. In *SIGMOD'13*.

[7] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien. Fast support for unstructured data processing: The unified automata processor. In *MICRO-48'15*.

[8] D. Ficara, S. Giordano, G. Procissi, et al. An improved DFA for fast regular expression matching. *ACM SIGCOMM*, 38(5):29–40, 2008.

[9] R. W. Floyd and J. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29(3), 1982.

[10] E. S. Fukuda, H. Inoue, T. Takenaka, et al. Caching memcached at reconfigurable network interface. In *FPL'14*.

[11] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with FPGAs. In *FCCM'13*.

[12] M. Hutton. Architectural paths to faster and more robust FPGAs. http://www.fpl2015.org/pdf/keynotes/3.pdf.

[13] Z. István, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *FCCM'16*.

[14] Z. Istvan, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *SIGMOD'14*.

[15] Y. Kaneta, S.-i. Minato, and H. Arimura. Fast bit-parallel matching for network and regular expressions. In *SPIRE'10*.

[16] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing GPU-accelerated Group-By and aggregation. In *ADMS'15*.

[17] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[18] C.-H. Lin, C.-H. Liu, and S.-C. Chang. Accelerating regular expression matching using hierarchical parallel machines on GPU. In *GLOBECOM'11*.

[19] F. Marass and C. Upton. Sequence searcher: A Java tool to perform regular expression and fuzzy searches of multiple DNA and protein sequences. *BMC research notes*, 2(1):14, 2009.

[20] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating snort ids. In *ANCS'07*.

[21] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.

[22] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Configurable hardware-based streaming architecture using online programmable-blocks. In *ICDE'15*.

[23] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *PVLDB*, 6(12):1310–1313, 2013.

[24] H. Noyes et al. Microns automata processor architecture: Reconfigurable and massively parallel automata processing. In *HEART'14*.

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

[26] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[27] H. Pirk. Efficient cross-device query processing. In *The VLDB PhD Workshop*, 2012.

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

[29] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA'14*.

[30] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA'99*.

[31] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen. Multi-query stream processing on FPGAs. In *ICDE'12*.

[32] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *FCCM'01*.

[33] E. Sitaridi, O. Polychroniou, and K. A. Ross. SIMD-accelerated regular expression matching. In *DAMON'16*.

[34] E. A. Sitaridi and K. A. Ross. GPU-accelerated string matching for database applications. *PVLDB*, 25(5):719–740, Oct. 2016.

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

[36] J. Teubner and L. Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[37] J. Teubner, L. Woods, and C. Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *SIGMOD'12*.

[38] W. S. Todd Mytkowicz, Madan Musuvathi. Data-parallel finite-state machines. ASPLOS'14.

[39] T. Ueda, M. Ito, and M. Ohara. A Dynamically Reconfigurable Equi-Joiner on FPGA. *IBM Tehnical Report RT0969*, 2015.

[40] L. Woods, Z. István, and G. Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 7(11), July 2014.

[41] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB*, 3(1-2):660–669, 2010.

[42] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*.

[43] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *ANCS'08*.