Optimising Operator Sets for Analytical Database Processing on FPGAs

Anna Drewes[1(✉)], Jan Moritz Joseph[1], Bala Gurumurthy[2], David Broneske[2], Gunter Saake[2], and Thilo Pionteck[1]

[1] Institute of Information Technology and Communications, Otto-von-Guericke University, 39106 Magdeburg, Germany
anna.drewes@ovgu.de

[2] Institute of Technical and Business Information Systems, Otto-von-Guericke University, 39106 Magdeburg, Germany

Abstract. The high throughput and partial reconfiguration capabilities of modern FPGAs make them an attractive hardware platform for query processing in analytical database systems using overlay architectures. The design of existing systems is often solely based on hardware characteristics and thus does not account for all requirements of the application. In this paper, we identify two design issues impeding system integration of low-level database operators for runtime-reconfigurable overlay architectures on FPGAs: First, the granularity of operator sets within each processing pipeline; Second, the mapping of query (sub-)graphs to complex hardware operators. We solve these issues by modeling them as variants of the subgraph isomorphism problem. Via optimised operator fusion guided by a heuristic we reduce the number of required reconfigurable regions between 30% and 85% for relevant TPC-H database benchmark queries. This increase in area efficiency is achieved without performance penalties. In 86% of iterations of the operator fusion process, the proposed heuristic finds optimal candidates, which is 3.6× more often than for a naive greedy approach.

Keywords: Query processing · Database operators · Operator fusion · Graph modeling · Heuristic · FPGA · Overlay architecture

1 Introduction

In order to address growing concerns regarding scaling and power efficiency, database research strives to include heterogeneous compute devices as accelerators [5]. In addition to a variety of GPU-based systems [3,13,30], FPGAs are also considered for their special capabilities: Their massive I/O-bandwidths, spatial parallelism, and deeply pipelined processing capabilities are all indicators for high performance in analytical database query processing. FPGAs have been

This work is funded by the German Research Foundation (DFG) projects PI-447/9 and SA-465/51-1.

F. Rincón et al. (Eds.): ARC 2020, LNCS 12083, pp. 30–44, 2020.
https://doi.org/10.1007/978-3-030-44534-8_3

used both as static accelerators for single database operations [2,12,18,24,27] and also as more general reconfigurable platforms [1,9,28,31]. These examples show large performance enhancements due to the use of heterogeneous hardware.

Despite the advantages of heterogeneous hardware, systems integration issues resulting from a bottom-up design process (cf. Fig. 1) impede their practical applicability. Thus, new optimisation techniques, operator representations, and memory management strategies have been introduced. Mapping queries to vastly differing processing architectures can be accomplished by converting SQL statements into sequences of simple, data-parallel operations [13]. Using this concept of primitive-based query processing, all operations besides pipeline breakers such as *Sort* can be evaluated as data flow graphs. For CPUs, evaluating these sequences of operations is possible via just-in-time compilation of optimised primitive implementations from intermediate representation (IR) [4,20]. This is not viable for FPGAs, as synthesis and place-and-route are very time-consuming compared to software compilation and can take from minutes to hours depending on the complexity and size of the target design. An alternative is sequential execution of pre-synthesised compute kernels, achieving high throughput through massive data parallelism. For FPGAs, implementing the basic database operations as OpenCL kernels follows this paradigm. In contrast to GPUs, where each kernel has access to the full hardware, the resources of an FPGA are either shared between a fixed set of static accelerators, or kernels have to be exchanged at runtime via costly reconfiguration of nearly the complete device [15,28,29]. Thus, both standard solutions for high performance integration of CPUs and GPUs are not targeting the special requirements of FPGAs.

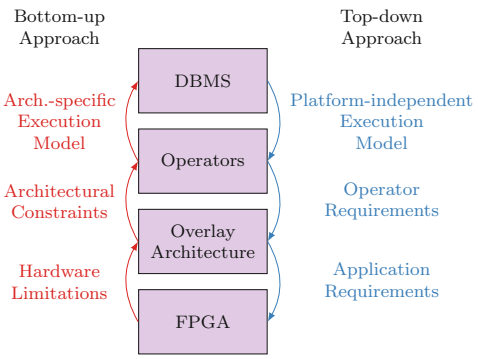

Fig. 1. Two standard approaches to system design/integration.

A better concept for FPGAs is an *overlay architecture*, where the logic resources of the FPGA are spatially divided into interconnected partitions which fit one hardware operator each. The exemplary design shown in Fig. 2 consists of 9 reconfigurable regions (RR) and a static partition, which is shown in blue and contains supporting logic such as bus systems, memory controllers, and the

host interface. Specially synthesized hardware operators (compute unit, CU) for data processing or interconnect bridges for data movement can be loaded into the reconfigurable regions via dynamic partial reconfiguration (DPR) of parts of the FPGA fabric. This is shown in red in Fig. 2. Thus, an overlay architecture is an abstraction of the raw FPGA into a set of interconnected compute units and exposes the spatial parallelism inherent to FPGAs in a practical way [6,31]. While this top-down design approach (cf. Fig. 1) solves the problem of synthesis at runtime, it leaves a whole new set of problems to be addressed. The main issue is that most of the database primitives require only a rather small amount of resources (cf. Sect. 5). At the same time even simple queries can require evaluation of a dozen operations in a single pipeline. This leads to the requirement that the FPGA has to be broken up into a very large number of small, but tightly interconnected, reconfigurable regions. This is highly resource-inefficient.

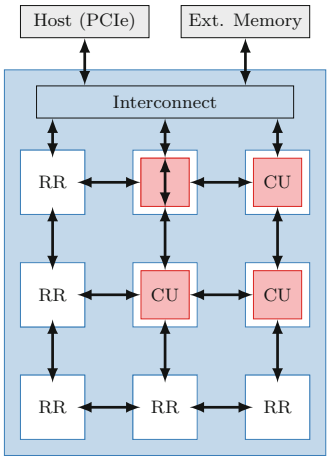

Fig. 2. Example of an FPGA overlay architecture. (Color figure online)

In order to address these urgent design issues, we propose an optimised operator fusion method for database primitives implemented for FPGAs targeting reconfigurable overlay architectures. We improve resource efficiency by reducing the number of reconfigurable regions occupied for executing a query. Specifically, we present an analysis of operators for analytical database query processing on FPGAs using overlay architectures. We reach an application-centered view of the design and integration process by formalising the apparent problems and reducing them to optimisation tasks based on known graph problems. This allows us to reduce the number of reconfigurable regions required for evaluating the considered analytical database queries by 52% on average, with only a slight increase RR size and most importantly, without performance penalties.

This paper is structured as follows: After discussing related work in Sect. 2, we provide an overview on primitives for analytical database query processing

for heterogeneous hardware in Sect. 3. Section 4 contains definitions and our proposed optimisation process for the two problems of operator granularity and matching of composed operators. Results for relevant benchmark queries are presented in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 Related Work

The problem addressed in this paper has no exactly matching related work where a direct comparison of results is possible in a meaningful way, but conceptually similar approaches with different optimisation criteria have been used in other application fields. Thus, in this section, we discuss related work on code fusion for heterogeneous hardware in general and optimisations regarding sets of operators for FPGAs.

Data-centric query compilation is important to achieve high performance on CPUs and GPUs [20]. While the problem of query compilation for CPUs and GPUs during runtime is structurally similar to our work, for FPGAs we have to focus on optimisations at design-time and can therefore improve and adapt not only the operator library, but also the static FPGA design. Menon et al. [19] describe a model for operator fusion during query planning in order to better exploit CPU caches. Apart from the lack of caches on FPGAs, their differing architecture requires custom models.

OpenCL can be used as a target platform for custom code generation for heterogeneous hardware. For example, Hawk [4] and Voodoo [22] generate optimised OpenCL code. These approaches not only cover code optimisations, but also address parallelism, both of which are orthogonal to the problems addressed in this paper, which deal with optimisation of the library of operators and reducing the number of discrete hardware operators required.

The model and optimisation approach for Code Fusion on GPUs presented by Wahib et al. [26] is, just as our approach, based on data dependencies between kernels. The authors propose a variation of genetic algorithms to solve the optimisation problem, while we propose a much more simple greedy constructive optimisation process. The results are not comparable to our results due to fundamental differences in optimisation goals: GPUs fuse kernels to reuse data fetched from memory, while we try to achieve better FPGA resource efficiency. Another, completely different problem for database operators on FPGAs involves fitting independent kernels into one large reconfigurable region covering most of the FPGA. Wang et al. [28] use dynamic programming to generate sequential query execution plans, and to partition sets of OpenCL kernels too large to fit into the singular reconfigurable region into several bitstream images based on a cost model. Since we build up queries from small, individually reconfigurable base primitives, this problem does not apply to our system.

Finally, RENO [21] is a data-dependency-driven optimiser implemented in hardware inside a CPU core, enabling the elimination or fusion of certain instructions directly from the instruction stream. Similarly, there exists work on hardware-based fusion of simple instructions into larger macro-operations in

order to allow for a more efficient CPU microarchitecture [7,17]. All three works employ mechanisms similar to our work, but focus on the design of efficient high performance hardware implementations of code fusion engines, which are unnecessary in our work, since we do not use instruction-driven processing elements, but assemble pipelines of specialised hardware operators.

3 Database Primitives

Using *database primitives* to split up queries into small highly optimised base operations enables high-performance query processing on heterogeneous hardware architectures. For FPGAs, mapping primitives to reconfigurable regions of an overlay architecture makes construction of deep hardware pipelines possible at runtime. As *database primitives* is a widely used term in various contexts, we begin with a brief introduction and definition of the usage of this term in the scope of this paper.

After a database management system (DBMS) has parsed and optimised a query submitted by a user, it generates a query execution plan. This is necessary since SQL is a declarative language, and thus only describes the desired result and not a sequence of operations [16]. While large parts of parsing and optimisation can be similar for different execution techniques, the actual query execution plan is inherently architecture-specific. One technique for evaluating analytical database queries in heterogeneous systems is the use of database primitives [11,13,22]. These primitives represent the basic operations of parallel programming and can thus be directly implemented on a variety of compute architectures.

We consider *five types of Atomic Primitives*, some of which can be composed to form more complex operations required for evaluating queries.

1. Using the *Map* primitive, a function is applied to all values of one or more input vectors producing an output vector with the same number of tuples. A special case is the pairing of a scalar with each element of a vector. Since there are no dependencies between individual vector elements, *Map* can be parallelised trivially.
2. A *Reduce* operation takes in a vector and aggregates all elements into a single scalar. This primitive can be parallelised using reduction tree approaches, both in coarse- and fine-grained fashions, depending on the target architecture. Analytical database query processing requires a specialised variant of this primitive: *Grouped Aggregation*, which, instead of performing one reduction over the entire input vector, uses another input vector of group IDs to split the data input vector into multiple ranges. This operation can also use a tree-like task structure for parallelisation [14,23].
3. The *Prefix Scan* primitive is mostly used to generate index vectors for other primitives. Thus, probably the most common operation is prefix sum. Parallelising *Prefix Scan* is challenging. A standard approach is to split the input data into multiple chunks, each processed independently. Then the singular result values are aggregated by a single task to generate offsets that can be applied to the chunks in a final step using *Map* [23].

4. *Scatter/Gather* do not provide computation by themselves, but instead rearrange data by using indexed reads or writes. These primitives can exhibit widely varying memory access patterns. The impact of highly parallel *Scatter/Gather* on the memory system has to be carefully considered during implementation.
5. *Sort* is used in various ways in database systems. While preprocessing parts of the input vector is possible, the whole input vector has to be available before any output can be computed. This means that *Sort* is a pipeline breaker. Thus, interaction with other primitives is limited.

Some database operations cannot be captured fully by any one of these categories. They may be described as *Composed Primitives*, consisting of base primitives and special additional functionalities. A common example is construction of hash tables: While the computation of the hash function(s) is described by *Map*, modeling the insertion process using the base primitives is not sensible. Since the behaviour of different hash table variants is highly dependent on data distribution, fusing these operations with other primitives may result in inflexible and slow designs or require a vastly higher number of pre-synthesised operations. Thus, both sorting and hash table-manipulation operations are not considered in this work.

Mapping operators to DPR regions at runtime is a considerable problem, that has been addressed by other works [6], and thus is not detailed in this paper.

4 Optimisation Targets

As already introduced, we achieve more resource-efficient database query processing on FPGAs by targeting two objectives: *Hardware Operator Granularity* optimisation reduces resource requirements by lowering the number of reconfigurable regions required for executing a query, while optimised *Matching of Composed Operators* allows for reduced operator library size and thus improves synthesis time and the required storage space for the operators. In the following subsections, we use standard graph approaches for these optimisation targets.

4.1 Hardware Operator Granularity

Our first optimisation target is hardware operator granularity. In detail, we will present hardware constraints, describe the problem model, introduce operator fusion, and propose a greedy optimisation process.

For the purpose of modeling, we adopt an abstraction of a generalised overlay architecture for FPGAs, where the FPGA is partitioned into a static design and dynamically reconfigurable regions. Memory controllers, as well as an interconnect and various management logic forms the static partition, while hardware operators can be loaded into reconfigurable areas at runtime. The FPGA is thus abstracted into a set of directly interconnected compute units. To keep the amount of constraints low, we assume that each operator can be mapped to

every reconfigurable region. For effective hardware pipeline processing, the links between reconfigurable regions are designed to stream data at the bandwidth required by the operators. Only data transfers to and from off-chip memory are assumed to be handled by DMA engines in the static partition.

We model query execution plans as directed, acyclic, coloured graphs $G = (V, E)$. Database primitives, as well as data sources and sinks form nodes $v \in V$. Their colouring (tag) is based on the underlying operator and data types. We define this colouring as *operators*. An operator node in G is defined as an *instance* and needs to be mapped to a reconfigurable region for execution. The directed edges $e \in E$ represent the data dependencies present in the query execution plan. Note that the size of data transfer that is represented by an edge may not necessarily be identical for all edges of a query, as in many cases the actual amount of data transferred is determined by characteristics of the input data. An example query execution plan generated from the TPC-H analytical database benchmark query Q6 is shown in Fig. 3 [25]. It contains a typical amount and complexity of operations, except for joins, which are explicitly not covered in this work. Data flows from bottom to top and the different data transfer sizes are highlighted at the edges. After evaluating the selection criteria via *Map less-than/greater-than (map-s-lt/map-s-gt)* comparison operations, the results are combined via *map-and*. Then, the *filter* operation eliminates all elements of the input vectors that do not meet the selection criteria. The filtered input columns are multiplied element-wise *(map-mul)*. Finally the result is computed by aggregating the result vector of the multiplication operation *(red-add)*.

In general, non-parallelised database primitives that are synthesised as streaming hardware operators have rather small resource requirements. This is especially relevant for runtime-reconfigurable implementations on FPGAs using a multitude of DPR regions, as there is a high degree of area overhead associated with each runtime-reconfigurable region. This overhead is due to the need to provide communications infrastructure and isolation of the reconfigurable region from the rest of the system during reconfiguration. This leads to the goal of fusing database primitives that commonly occur together into larger units, thus reducing the number of DPR regions required for evaluating a query. While the base query Q6 shown in Fig. 3 would require 15 distinct hardware operators, the optimised query shown in Fig. 6 reduces the number of required DPR regions to 4 (cf. Table 1). Operator fusion also relieves some amount of load from the interconnect of the overlay architecture because in many cases fused primitives share input vectors. Data transferred between fused operators are removed from the system interconnect as well. In addition, area efficiency is improved in general, as scheduling can be optimised by the design tools if the operators are synthesized in a fixed combination inside a single reconfigurable region. Of course, the DPR regions have to be sized according to the single largest primitive or fused operator. As FPGA synthesis and place-and-route are time-consuming and resource-intensive tasks, it is generally not feasible for a database management system to synthesize custom hardware accelerators for each query at runtime. Thus, there remains the problem of deciding which combinations of primitives

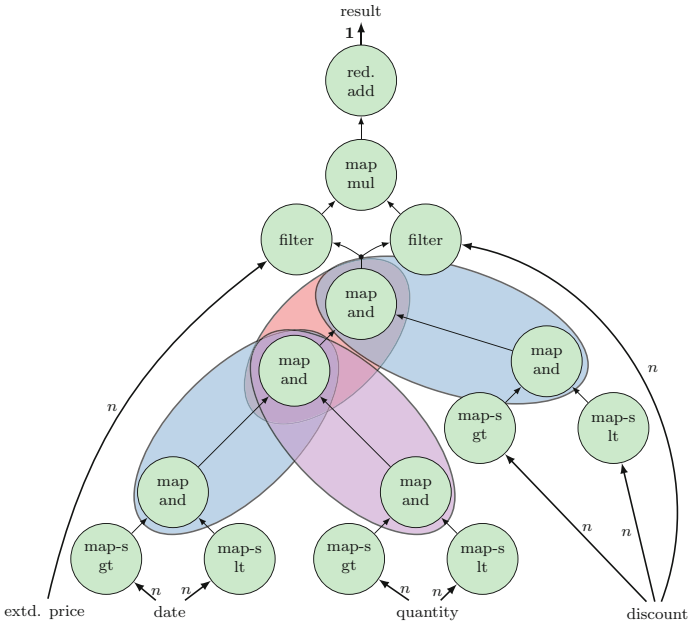

Fig. 3. Query execution plan of TPC-H Query 6. (Color figure online)

should be fused into composed operators. These decisions are relevant, as the size of the operator library is limited by both the available storage space and the time overhead at runtime for finding fitting operators.

In order to optimise the set of required primitives to generate fused operators, we start with a set of query execution plans S_{base}. This set can be either extracted from logs or traces of a running system or generated from database benchmark suites. S_{base} induces a set of operators O_{base}, which describes the set of database primitives used across all queries of S_{base}. The formal goal is to identify the maximum common induced subgraph J for each possible tuple of these graphs (G, H) with $G \in S_{\text{base}}, H \in S_{\text{base}}$ and $G \neq H$. Since the decision problem whether two graphs G, H have a common subgraph J of size k is NP-complete [10], in practice this problem cannot be expected to be solved in an optimal way.

We propose a greedy, constructive algorithm for generating fused operators from a base set of queries. Our iterative process can be summarised as follows: First, we identify the most common combination of operators $(a, b) \in O_i \times O_i$, where O_i is the induced set of operators O after iteration i with $O_0 = O_{\text{base}}$. Second, fuse the identified fusion candidate into a new operator $o = (a, b)$. Third, generate query graphs $G' \in S_{i+1}$ from the graphs $G \in S_i$ by replacing instances of (a, b) with o. This also updates the induced set of operators O_i to the updated set O_{i+1} used in the next iteration. This process can continue until all possible primitives have been fused.

The main problem with this algorithm is the fact that the most commonly occurring operator combination may not be the operator combination that can be replaced most often. This happens because primitives in query execution plans graphs are often arranged in tree or chain-like shapes, thus influencing their neighbours. This is illustrated in Fig. 3: Fusion candidates consisting of two *map-and* operations can be found four times within the graph. They are highlighted using the red, violet, and blue ellipses. As the ellipses are overlapping each other, the fused operator cannot be instantiated four times. It is only possible to replace the right blue ellipse and either the violet or the other blue one resulting in two uses of the fused operator. If the red ellipse were to be replaced, all other possible instances of the fused operator would be blocked. Generalising this example, we conclude that the number of coloured edges, which describe an operator fusion candidate, does not indicate the expected optimisation potential adequately. Solving this problem optimally requires frequent identification of maximum independent edge sets, or matchings. In order to avoid expensive calculation of such an optimal solution for every existing combination of operators in every iteration of the operator fusion process, we introduce a heuristic to select operator fusion candidates. Instead of the naive approach of using the number of edges of each colour to select fusion candidates, we propose to count the number of distinct nodes attached to edges of each colour. Divided by two, this heuristic provides a direct estimate of the target function, i.e. the number of expected possible instances of the potential fused operator. We evaluate the proposed heuristic in Sect. 5.

4.2 Matching of Composed Operators

At runtime, the problem of actually using the previously generated fused operators remains. This problem is especially relevant if the DBMS front end and logical optimisation stages are architecture-oblivious in order to support heterogeneous hardware other than overlay architectures of FPGAs. In this case an FPGA-specific optimiser has to identify (match) subgraphs H describing fused operators in an input query execution plan G, thus needing to repeatedly solve the subgraph isomorphism problem, which is NP-complete [8].

We suggest an approximation by following a greedy steepest gradient descent approach: Since G is a directed acyclic graph, match the largest fitting composed operator following the topological ordered query execution plan. After replacing the matched subgraph with the fused operator, the next matching is constructed.

In addition to the remaining high complexity of the approximation algorithm, sometimes the most appropriate fused operator is not an exact match. An example can be taken from the benchmark queries used in Sect. 5 and is shown in Fig. 4. This is a sequence of two primitives that commonly occurs as last processing step of queries. Data are aggregated according to a vector of group IDs (*gagg-add*). In addition to the sum, this sequence also computes an average value for each group (*map-div*). This requires information about the sizes of the distinct groups (*count*). In general, it is only necessary to count the

group sizes once. In contrast, duplicating the *gagg-count* operator for each composed grouped aggregation operation, as depicted in Fig. 5, eliminates the input vector *count* from the fused operator. Since this additional fused primitive is not complex, this is a better optimisation of the query execution plan, but much more difficult to realise, as it requires a fuzzy approach to subgraph matching.

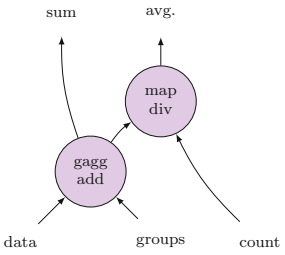

Fig. 4. Default replacement candidate for exact matching.

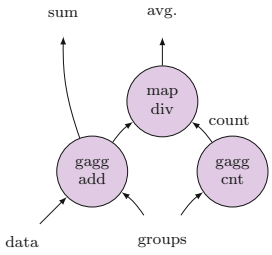

Fig. 5. Candidate for fuzzy matching.

5 Results

To evaluate our proposed optimisation process and heuristic, we applied our algorithm to relevant analytical benchmark queries from TPC-H [25], and also synthesized both the required primitives and fused operators.

5.1 Evaluation Setup

Table 1 lists the considered queries. As sorting and table join operations are pipeline-breaking operators and require implementations tailored to specific data distributions, fusing them will result in either inflexible operators or a large operator library. Thus, we do not consider those parts of the queries. Due to pipeline breakers within their query execution plan, many queries cannot be evaluated using a single continuous processing pipeline. These queries have to be split up into multiple processing pipelines. The partial query execution plans tagged *begin* contain the first operations, usually applying of selection criteria. The partial query execution plans tagged *end* cover final processing steps, such as grouped aggregation. The query execution plans are generated from the SQL statements using standard textbook concepts.

All primitives and optimised fused operators are written in C as single-tuple-per-cycle compute kernels with a data width of 32 bit. They are synthesised using Vivado HLS 2018.1. The directives HLS_DATAFLOW and HLS_PIPELINE are used to generate efficient streaming compute components. Throughout the whole project AXI Stream ports are used. We try to achieve the highest possible clock rate while maintaining a loop initiation interval of one clock cycle, with a starting frequency of 300 MHz. Operators failing to meet this requirement are synthesised

Table 1. Results

Query	Throughput GB s^{-1}		Reconf. Regions (RR Instances)			Resources				
						CLB		DSP		Savings
	Base	Opt.	Base	Opt.	Savings	Base	Opt.	Base	Opt.	
Single RR			1	1		400	450	10	10	–12%
Q1 (begin)	2.30	2.30	4	2	50%	1600	900	40	20	44%
Q1 (end)	14.33	14.33	13	9	30%	5200	4050	130	90	22%
Q4 (begin)	4.60	4.60	6	4	33%	2400	1800	60	40	25%
Q6 (complete)	–	4.60	15	4	73%	6000	1800	150	40	70%
Q12 (begin)	5.75	5.75	12	6	50%	4800	2700	120	60	44%
Q12 (end)	4.41	4.41	3	4	25%	2400	1800	60	40	25%
Q16 (begin)	–	4.60	20	6	70%	8000	2700	200	60	66%
Q19 (complete)	–	9.20	76	12	**84%**	30400	5400	760	120	**82%**
Average improvement					**52%**					**47%**

at lower frequencies. Only the division and grouped aggregation operators require a lowered clock frequency of 250 MHz.

The Xilinx Zynq ZC706 evaluation board is used as the design target. The FPGA part of the Zynq 7Z045 system-on-chip (SoC) has access to 2 GiB of external DDR3 memory with a maximal theoretical bandwidth of 19.2 GB s^{-1}. Its FPGA resources comprise lookup tables (LUT) and flipflops (FF) for implementing logic circuits, which are combined into Configurable Logic Blocks (CLB) that are tiled across the FPGA fabric. The FPGA also contains multiply-accumulate units (DSP) to simplify implementation of certain arithmetic operations. The considered FPGA contains a total of 27 325 CLBs and 900 DSP slices. The reconfiguration granularity for logic resources is a column of 50 CLBs, each consisting of 2 slices of 4 LUTs, while the reconfiguration granularity for DSP slices is 10. Therefore, the size of 2489 LUTs and 2950 FFs of the division operator mandates a minimum reconfigurable region size of 8 columns, which corresponds to 400 CLBs, or 3200 LUTs. Due to the layout of the clock regions within the FPGA fabric, up to 14 completely independent reconfigurable regions can be instantiated. Please note that only those queries with a small number of primitives can be processed in a single compute pipeline in the baseline.

5.2 Optimisation Process

Running the optimisation process until all reused operator combinations were found generates 12 fused operators within 21 fusion steps. The optimisation process is terminated when there are no more duplicated subgraphs. The average fused operator is composed from five base primitives, while the smallest fused operators contains only two. Because TPC-H Query 19 makes use of an identical set of extensive selection criteria three times, the largest fused operator maps to one of these sub-trees and consists of 23 primitives. As an illustrative example,

Fig. 6 shows the optimised query execution plan for the baseline query as shown in Fig. 3 utilising the fused operators shown in Figs. 7, 8, and 9. The primitives and fused operators are highlighted in violet, while the base primitives are shown in green. Figure 4 shows another example of a fused operator.

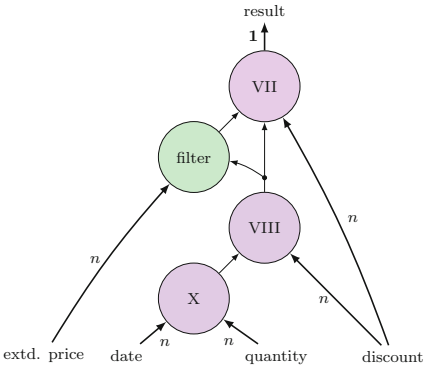

Fig. 6. TPC-H Query 6 using fused operators. (Color figure online)

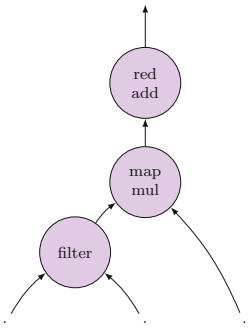

Fig. 7. Fused operator VII. (Color figure online)

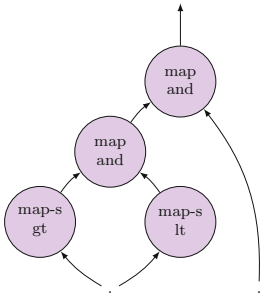

Fig. 8. Fused operator VIII. (Color figure online)

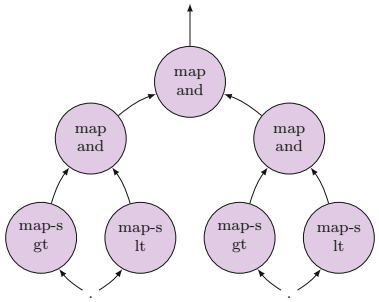

Fig. 9. Fused operator X. (Color figure online)

5.3 Discussion

In order to test the accuracy of our heuristic, we also computed optimal results. Our proposed heuristic correctly identified optimal fusion candidates in 18 out of 21 iterations of our algorithm. This estimation accuracy of 86% is over three times larger than the accuracy of the naive approach.

The reductions in resource requirements achieved by our operator fusion optimisation process are illustrated in Fig. 10: Fusing small database primitives significantly improves resource efficiency for overlay architectures in database query

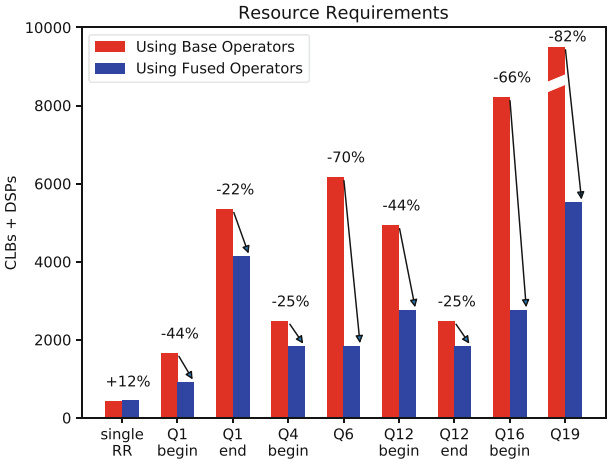

Fig. 10. Comparison of resources requirements using base primitives and fused operators for relevant TPC-H benchmark queries.

processing on FPGAs. Resource requirements for complete queries are reduced by up to 82%. This is an impressive result that shows that FPGA-enabled implementations of primitive-based analytical database management systems require architecture-specific optimisation of operators at design time. As the largest primitive, namely division, is part of the fused operator shown in Fig. 4, the size of each reconfigurable region increases slightly to 450 CLBs, as is shown in the first line of Table 1. This increase in size reduces the savings achieved by use of the fused operators, as can be seen when comparing columns six and eleven of Table 1. On average, the number of reconfigurable regions used for a single query is reduced by 52%, which leads to an average reduction in resource requirements of 47%.

There is no immediate impact on performance, as the maximum clock speed of the operators was not reduced by the fusion process, but due to the lowered number of reconfigurable regions, all queries can now be executed in a single pass. In contrast, the reconfiguration time between queries is reduced, as even with the slightly larger partitions, each query requires a significantly lower number of (re-)configured tiles. The throughput numbers shown in Table 1 refers to the total amount of data being read into and written out of the processing pipeline. Also, as communication between primitives within one fused operator is handled entirely within the operator, the interconnect in the static partition is relieved.

Due to the significant improvement in resource efficiency, our proposed approach improves the feasibility of integration of FPGAs using overlay architectures into analytical database systems. Our approach can also be applied to other application areas, such as network traffic analysis, digital signal processing, and software-defined radio, where data flow graphs generated at runtime are processed using a fixed library of operators.

6 Conclusion

In this paper we analyse the problem of automated fusion of commonly occurring combinations of basic database operators for FPGAs and propose solutions based on standard graph problems. While the underlying problem of identifying commonly occurring subgraphs in a set of graphs is very difficult and in the general case even hard to approximate, the special instance discussed here allowed for good approximation results after developing a heuristic. For the set of standard benchmark queries considered, the proposed optimisation process reduces the number of required reconfigurable regions by about 52% on average with a maximum reduction of 84%. As the size of each reconfigurable region increases slightly due to fusing of operators, the overall resource savings are smaller: Executing a query using fused operators requires between 22% and 82% less FPGA resources than using base primitives. The proposed optimised operator fusion process enables practical applicability of FPGA-based accelerators in query processing, thus increasing efficiency in handling large-scale database systems. Further constraining of the search space may be an interesting direction for future research.

References

1. Backasch, R., Hempel, G., Pionteck, T., Groppe, S., Werner, S.: An architectural template for composing application specific datapaths at runtime. In: ReConFig (2015)
2. Becher, A., Ziener, D., Meyer-Wegener, K., Teich, J.: A co-design approach for accelerated SQL query processing via FPGA-based data filtering. In: FPT, pp. 192–195 (2015)
3. Breß, S., Heimel, M., Saecker, M., Köcher, B., Markl, V., Saake, G.: Ocelot/HyPE: optimized data processing on heterogeneous hardware. PVLDB **7**(13), 1609–1612 (2014)
4. Breß, S., Köcher, B., Funke, H., Zeuch, S., Rabl, T., Markl, V.: Generating custom code for efficient query execution on heterogeneous processors. VLDB J. **27**(6), 797–822 (2018). https://doi.org/10.1007/s00778-018-0512-y
5. Broneske, D., Breß, S., Heimel, M., Saake, G.: Toward hardware-sensitive database operations. In: EDBT, pp. 229–234 (2014)
6. Capalija, D., Abdelrahman, T.S.: A high-performance overlay architecture for pipelined execution of data flow graphs. In: FPL, pp. 1–8 (2013)
7. Celio, C., Dabbelt, P., Patterson, D.A., Asanovic, K.: The renewed case for the reduced instruction set computer: avoiding ISA bloat with macro-op fusion for RISC-V. CoRR abs/1607.02318 (2016). http://arxiv.org/abs/1607.02318
8. Cook, S.A.: The complexity of theorem-proving procedures. In: ACM STOC, pp. 151–158 (1971)
9. Dennl, C., Ziener, D., Teich, J.: On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library. In: FCCM, pp. 45–52 (2012)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)

11. Gurumurthy, B., Broneske, D., Drewes, T., Pionteck, T., Saake, G.: Cooking DBMS operations using granular primitives - an overview on a primitive-based RDBMS query evaluation. Datenbank-Spektrum **18**(3), 183–193 (2018). https://doi.org/10.1007/s13222-018-0295-8
12. Halstead, R.J., et al.: Accelerating join operation for relational databases with FPGAs. In: FCCM, pp. 17–20 (2013)
13. He, B., et al.: Relational query coprocessing on graphics processors. ACM TODS **34**(4), 21:1–21:39 (2009)
14. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. PVLDB **6**(9), 709–720 (2013)
15. Intel Corp.: Intel FPGA SDK for OpenCL Programming Guide (2017)
16. International Organization for Standardisation: ISO/IEC 9075 Information Technology - Database Languages - SQL (2016)
17. Kim, I., Lipasti, M.H.: Macro-op scheduling: relaxing scheduling loop constraints. In: MICRO, pp. 277–290 (2003)
18. Koch, D., Tørresen, J.: FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In: ACM SIGDA, pp. 45–54 (2011)
19. Menon, P., Pavlo, A., Mowry, T.C.: Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. PVLDB **11**(1), 1–13 (2017)
20. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB **4**(9), 539–550 (2011)
21. Petric, V., Sha, T., Roth, A.: RENO - a rename-based instruction optimizer. In: ISCA, pp. 98–109 (2005)
22. Pirk, H., Moll, O., Zaharia, M., Madden, S.: Voodoo - a vector algebra for portable database performance on modern hardware. PVLDB **9**(14), 1707–1718 (2016)
23. Roosta, S.H.: Parallel processing and parallel algorithms - theory and computation. Springer (2000). http://www.springer.com/computer/swe/book/978-0-387-98716-3
24. Teubner, J., Woods, L.: Data Processing on FPGAs. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, San Rafael (2013)
25. Transaction Processing Performance Council (TPC): TPC BENCHMARK H (Decision Support) Standard Specification (2017)
26. Wahib, M., Maruyama, N.: Scalable kernel fusion for memory-bound GPU applications. In: SC, pp. 191–202 (2014)
27. Wang, Z., He, B., Zhang, W.: A study of data partitioning on OpenCL-based FPGAs. In: FPL, pp. 1–8 (2015)
28. Wang, Z., Paul, J., Cheah, H.Y., He, B., Zhang, W.: Relational query processing on OpenCL-based FPGAs. In: FPL, pp. 1–10 (2016)
29. Xilinx Inc: SDAccel Development Environment User Guide (2016)
30. Zhang, S., He, J., He, B., Lu, M.: OmniDB: towards portable and efficient query processing on parallel CPU/GPU architectures. PVLDB **6**(12), 1374–1377 (2013)
31. Ziener, D., et al.: FPGA-based dynamically reconfigurable SQL query processing. ACM TRETS **9**(4), 25:1–25:24 (2016)