Resource Elastic Database Acceleration

Resource Elastic Database Acceleration

Kristiyan Manev, Dirk Koch
Department of Computer Science, The University of Manchester, Manchester, UK
Email: {kristiyan.manev, dirk.koch}@manchester.ac.uk

***Abstract*—Database sizes are growing faster than the processing power in the post-Moore era due to the advent of big data applications, which make hardware acceleration mandatory. We propose dynamic stream processing using partial reconfiguration to provide a high-performance solution to runtime-known problems. This work researches the benefits of applying resource elastic techniques when building the execution pipeline.**

I. Introduction

In the big data era, the need for high-performance computing is ever-increasing and conventional processing is struggling to keep up with the exploding computational needs. FPGA technology is continuously evolving and recent generations provide millions of Look-Up-Tables (LUTs) and thousands of Digital Signal Processors (DSPs). As a traditional big-data problem, database acceleration has been targeted for FPGA acceleration for decades [1]. However, the reconfigurability of FPGAs is usually not utilized. Proposed database accelerators can be grouped into three main categories considering the flexibility to execute runtime-known problems: 1) Static, 2) Parameterizable Static, 3) Dynamic.

Static database solutions require the conversion of a query to RTL description. This process is done either manually or by utilizing HLS. This approach can achieve high throughput and logic efficiency. However, it requires that all query parameters are hard-coded prior to synthesis. Long synthesis times (up to several hours) combined with the requirement for resynthesis upon query parameters modification drives this approach unsuitable for most real-world database applications.

Alternatively, runtime initializable registers can be added to statically synthesized systems for database acceleration [2]. They allow some query modifications without the requirement for hardware resynthesis. However, placing parameters in registers disables some optimisations that synthesize tools can exploit with hard-coded values, thus resource requirements are generally increased. Additionally, such systems require the presence of all PEs that have to possibly be used for query acceleration. Targeting this approach can lead to: 1) inability for query acceleration, due to insufficient amount of PEs to handle complex subquery requirement, 2) overprovisioning of PEs that are not used in (most) queries.

Recent works utilize reconfigurability in FPGAs to allow dynamic database acceleration [3]. Building the execution pipeline at runtime allows parameterized PE solution without the drawbacks of overprovisioning.

This project aims to deploy a system that utilizes a library of parameterizable PEs that are placed in a partially reconfigurable region at runtime to build the execution pipeline (see Fig. 1). Also, applying resource elastic techniques will allow alternative configurations, which can further elevate the achieved execution efficiency and reduce execution time.

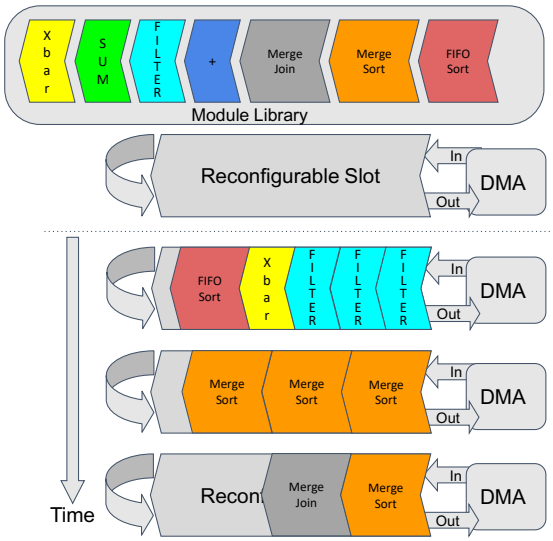

Fig. 1: Stream processing example of module placement in space and time to execute a database query

II. Resource Elastic Stream Processing

A. Overview

Ideally, our system will map queries at runtime with high scheduling efficiency. This is achieved by translating received queries for acceleration into graph representation of the accelerator modules needed. A scheduler maps nodes from this graph to PEs from our library modules (provided as relocatable configuration bitstreams), which creates an intermediate graph representing the dataflow between PEs. Some operations can have multiple module variants and thus we have different scheduling plans depending on runtime requirements and available resources. Using post-P&R relocation [4] we can stitch the bitstreams of modules producing a configuration that occupies a full reconfigurable slot. For a full Ultra96 slot, relocation takes $\sim 13ms$ and bitstream configuration $\sim 6ms$.

To enable resource elastic capabilities to full extend, we define a custom interface between modules that allows modules to communicate by streaming data between them on virtual channels. Our custom interface is an updated version of the one described in [5]. Additions to the interface specification include: 1) fields for sharing data intermediate values between modules, 2) protocol for modules to have direct access to external memory, 3) protocol for modules to have memory mapped registers in a global address space.

Size	Table 'part'	Table 'lineitem'
Initial	2,000,000	59,986,052
After pre-filter	1,284,812	4,754
After join	3,077	
After final filter	1,133	

TABLE I: Amount of tuples for TPC-H SF=1 Q19

B. Resource Elasticity

Resource elasticity is the ability to trade-off between hardware resources and achieved throughput or utility (we define utility as the amount of work performed per unit of I/O). This can be accomplished by replicating a PE multiple times or by replacing the module with an alternative one that has different resource constraints.

To present the benefits of resource elasticity we consider TPC-H [6] Query 19 due to its complex data filtering requirements. Two execution approaches are available: 1) join→filter, 2) partial-filter→join→filter. The second approach requires additional filtering resources but the data filtering prior to expensive operations (in this case a table join) commonly will result in overall faster execution. We build a TPC-H database with SF=1 and Table I shows that pre-filtering tables results in almost 98% of the tuples to be filtered out prior to join.

Filter modules (where clauses) that use Disjunctive Normal Form (DNF) representation for Boolean evaluation targeting Xilinx UltraScale+ are presented in [5]. By providing three generic filter modules (DNF8, DNF16, DNF32), all filter operations can be implemented with little resource overhead for arbitrary resource requirements. The filter operation is present three times in the execution graph with different parameters. We evaluate some of the possible module placements for each of the three operations in Table II. The results show resources as the amount of LUTs as well as in terms of resource columns (as described in [5]). Some configurations require data duplication in order to perform many comparisons on certain data fields. Thus we show throughput as a percentage of the peak datapath throughput. For instance, a system utilizing 512-bit datapath at 300MHz provides a peak throughput of 19.2 GB/s, which would ideally correspond to the available 100% throughput. Results show that the throughput tends to scale sublinearly with allocated resources and achieving 100% peak throughput can be costly in resources. However, allowing a runtime system to take runtime resource allocation decisions will likely result in overall better performance.

C. Discussion

Multiple factors can influence the system scheduler in order to apply various trade-offs. For example, one main factor is the amount of data transferred for a particular computation. E.g., a small database table will have a short runtime even when running on a slow PE, thus hardware resources can potentially be reallocated to other PEs. Factors can also be dependant on the particular operation performed. For example, merge sorters can have various utility and the total runtime does not exclusively depend on the merging throughput [7]. When scheduling a merge sort operation, a system will know the number of presorted streams prior to module placement, thus can take decisions (that may be counter-intuitive) that result in faster execution.

	Modules	Columns	LUTs	Throughput
Table 'part' pre-filter				
#1	3x(DNF32)	3x5	20,064	100%
#2	1x(DNF32)	5	6,688	57%
#3	1x(DNF16)	3	3,405	28%
#4	1x(DNF8)	2	2,114	16%
Table 'lineitem' pre-filter				
#1	1x(DNF16)	3	3,405	100%
#2	1x(DNF8)	2	2,114	77%
Final filter				
#1	1x(DNF32)	5	6,688	100%
#2	1x(DNF16)	3	3,405	83%
#3	1x(DNF8)	2	2,114	63%

TABLE II: Post - Place & Route utilization (as resource columns and LUTs) and throughput (as ratio to peak datapath throughput) of the evaluated module configurations.

III. Proposed and Future Work

The overall project goal is to utilize partial reconfiguration to enable a resource elastic runtime system to trade resources for throughput and to optimize the usage of the FPGA resources for problems only known at runtime. Future work includes integration of the presented system with an existing software Database Management System (DBMS) and also developing automated scheduler that can perform design space exploration on such resource elastic system and make optimal scheduling decisions as well as an evaluation of the approach.

IV. Conclusion

Previous works enable dynamic stream processing targeting big data problems such as database query acceleration. We anticipate that applying resource elasticity enables trade-offs in such systems that can eventually result in various scheduling options. Utilizing this technique allows schedulers to make smart decisions about the allocation of physical resources for different PEs and the expected runtime throughput.

References

[1] Philippos Papaphilippou et al., "Accelerating Database Systems using FPGAs: A Survey," in *FPL*. IEEE, 2018.

[2] S. Behzad et al., "AxleDB: A novel programmable query processing platform on FPGA," *Microprocessors and Microsystems*, vol. 51, 2017.

[3] Daniel Ziener et al., "FPGA-based Dynamically Reconfigurable SQL Query Processing," *TRETS*, 2016.

[4] K. Pham et al., "BITMAN: A Tool and API for FPGA Bitstream Manipulations," in *DATE*. IEEE, 2017.

[5] K. Manev et al., "Scalable Filtering Modules for Database Acceleration on FPGAs," in *HEART*, 2019.

[6] Meikel Poess et al., "New TPC Benchmarks for Decision Support and Web Commerce," *ACM Sigmod Record*, 2000.

[7] K. Manev et al., "Large Utility Sorting on FPGAs," in *FPT*, 2018.