A Relational Algebra Processor

6.375 Final Project Report

Ming Liu **Shuotao Xu**

Group 12

May 15, 2013

1. Motivation

Today's Database Management Systems (DBMS) are typically software running on a top of a standard operating system on a general purpose processor. Upon receiving a complex query statement (in SQL for example), the DBMS produces a query plan fundamentally based on relational algebra primitives, which is executed on the CPU. However, many such DBMS's are being used in the realm of scientific computing or analytics, where the data is read-intensive and queries are computationally heavy. For these types of workloads, the performance of a DBMS is often bottlenecked by processing power, software overhead, latency and power consumption.

We propose a Field Programmable Gate Array (FPGA) based relational algebra processor to compute database queries. At the system level, such a dedicated processor may be inserted between physical storage (e.g. SSDs or HDDs) and the host machine to directly process the queries where the data is located with minimal software intervention (Figure 1). Dedicated relational algebra operators are programmed on the FPGA to accelerate complex queries, exploiting parallelism and high speeds of dedicated hardware, while saving power and reducing bandwidth between the physical storage and host.

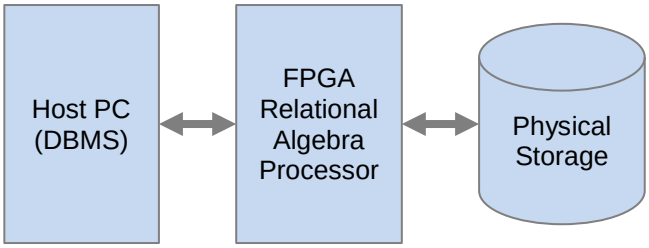

Figure 1: RA processor with direct access to storage

To narrow the scope of the project, we used on-board DRAM to emulate physical storage where the database tables reside (Figure 2). In addition, we did not run a full-fledged DBMS on the host system nor do we execute SQL queries. We considered only basic relational algebra operators, which forms a subset of the capabilities of SQL. Tables are read-only.

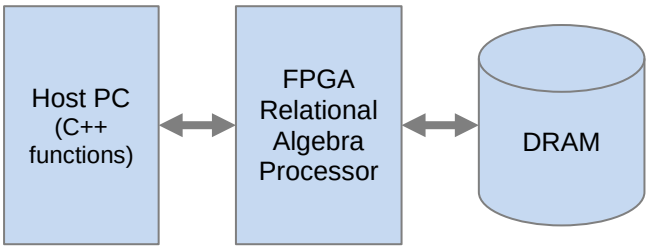

Figure 2: Proposed system with DRAM emulating storage

Relational algebra has five primitive operators: selection, projection, Cartesian product, set union and set difference. Those operate on relational database tables where data records are often referred to as rows, and record attributes are called columns.

1. Selection is a unary operation written as $\sigma_{\varphi}(R)$ where φ is a Boolean formula (>, =, < etc.) that consists of a set of conditions on the attributes of relational database table R. Selection outputs all those tuples in R for which φ holds.
2. Projection is a unary operation written as $\pi_{a_1,\ldots,a_n}(R)$ where $a_1, \ldots, a_n$is a set of attribute names (columns) of the relational database table R. Projection outputs only those attributes of the table and also eliminates duplicated outputs.

3. Cartesian product is a binary operation written as $R \times S$, where R is a m-tuple table of a rows and S is n-tuple table of b rows. The result is a $m + n$ tuple table of $a \times b$ rows.
4. Set Union is a binary operation written as $R1 \cup R2$, where $R1$ and $R2$ are the relational database tables of the same schema, (i.e. same columns). This results $R = \{t | t \in R1 \vee t \in R2\}$
5. Set Difference is a binary operation written as $R1 - R2$, where $R1$ and $R2$ are the relational database tables of the same schema, (i.e. same columns). This results $R = \{t | t \in R1 \vee t \notin R2\}$

Altogether, the five primitive operators have the expressive power to derive many other relational algebra. For example, a JOIN in SQL $R \bowtie_{\varphi} S$ could be expressed as $\sigma_{\varphi}(R \times S)$.

2. Project Objective

The goal of this project is to develop a stand-alone relational algebra processor on the FPGA and to explore the type of queries that are advantageous for offloading execution from the host to the FPGA. The system will accept as input a set of tables representing a database and a series of fundamental relational algebra operations. It will perform the operations on the FPGA and output a single table containing the results.

Our secondary objective is to outperform a lightweight database management system, SQLite, in execution time on the same set of queries on the same tables running on a conventional computer. We hypothesize, however, that SQLite will remain the faster option in data I/O heavy queries due to high memory bandwidth on modern computers, whereas our FPGA RA processor will surpass SQLite in compute heavy queries.

For simplicity and ease of implementation, some restrictions are imposed on the input data:

- Table types are 32-bit integers/chars only
- The size of all database tables do not exceed the size of DRAM
- Max number of columns per table is 32
- Max number of predicates of SELECT is 16

3. High-level Design and Test Plan

At the top level, the system is composed of software running on the host computer, which parses input tables and queries, streams them over PCIe to the FPGA which processes them. Result tables are then streamed back over PCIe to the host which finally displays the output table on the screen.

3.1 Top Level Hardware and Software Architecture

Figure 3 shows the high level hardware system architecture of the relational algebra processor on the FPGA. Overall, it consists of:

- 6 RA Operators to perform basic relational algebra operations
- Data I/O block to send data between the host and the FPGA via PCIe
- RA Controller to coordinate the operation of the relational algebra commands
- DRAM Controller to load/store data into DRAM
- Row Marshaller to translate table specifications into raw DRAM address/data

All of the RA Operators blocks are fully connected to each other as well as to the Row Marshaller. Data may be streamed in or out via any of the operator blocks and passed between blocks for sequential processing without storing intermediate values.

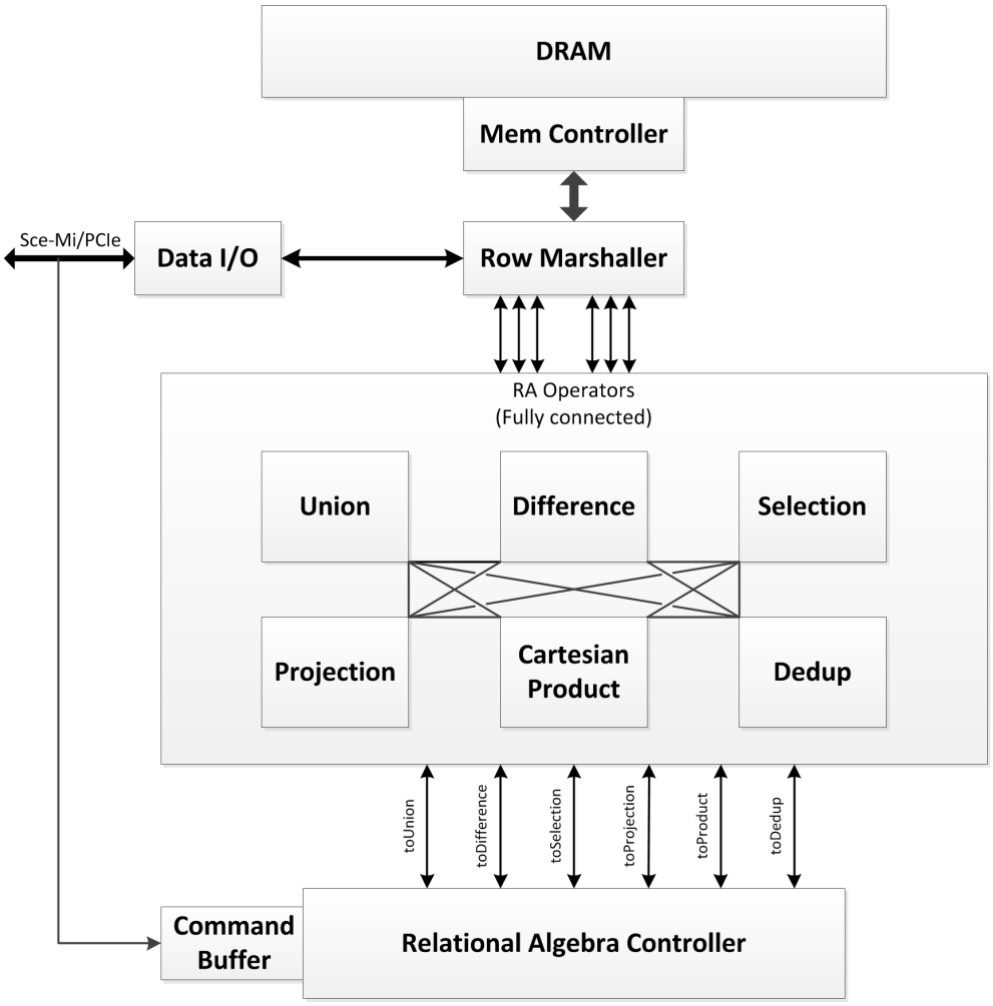

Figure 3: Top level hardware architecture diagram

Upon starting the system, the software first streams the entire set of database tables into DRAM via SceMi/PCIe, and software fills a Command Buffer which specifies the series of relational algebra operations to be performed on these tables. The RA Controller coordinates the execution of these commands by enabling the necessary RA Operator blocks in sequence and intelligently directing the flow of data among the RA Operator blocks. The results are stored back into DRAM and then streamed out to the host PC via SceMi/PCIe. Note that it is not always possible to bypass storing intermediate values, and the RA Controller coordinates the blocks to use DRAM as scratch space.

Query scheduling and metadata management for each table, such as their address in DRAM, size and properties, are managed in software for simplicity. The necessary metadata is passed to the hardware as part of the command so that hardware knows how to handle the tables. The overhead of doing this in software is small compared to the time it takes to actually process the tables because it is only done once at the start of a new query.

3.2 Test Plan

Testing was done at the system level and the block level. At the system level, the RA processor was tested for correct functionality against a conventional DBMS, SQLite, by comparing the outputs of the RA processor with the outputs from SQLite given the same queries. We initially populated the FPGA DRAM and SQLite with the same tables of various sizes and attributes. Then sets of relational algebra queries were executed:

- single operator queries: basic test for each of the operators
- complex operator queries: permutations of multiple operators one after the other, with data dependencies

Note that SQLite only accepts SQL queries and not relational algebra operators, thus conversion is required. We used an open source RA interpreter [1], which accepts RA operators and directly issues the translated queries to SQLite.

Separate test bench environments were created for the Row Marshaller and each RA operator module. A DDR2 model was used in all of the test benches. The Row Marshaller was tested by writing table data and then verifying that the same data can be read back from any of its access ports. The RA operator test benches composed of the operator module under test, the Row Marshaller and the DDR2 model. Commands are issued to the operator and we verify that upon receiving the acknowledgement from the operator that the expected output table resides in DRAM by reading it out via an access port of the Row Marshaller.

4. Microarchitecture Description

Refer to Figure 3 for the top level diagram of the system.

4.1 Host Software

Software running on the host machine is responsible for streaming table data to and from the FPGA DRAM, managing table metadata and packing/optimizing relational algebra commands to be sent to the FPGA. Refer to Figure 4 for the software block diagram.

Database tables, stored as CSV files on the host, are tokenized by the software. A separate table metadata entry is created for each table and kept in a global structure on the host. Table contents are streamed over SceMi ports via PCIe to populate the FPGA DRAM. Another CSV file containing the relational algebra commands is also parsed, and the commands are packed into structs. The software then precomputes the metadata of the output table (assuming worst case sizes) such that any dependent commands will have the correct metadata. The commands are reordered and scheduled (algorithm is discussed later), and then passed to the FPGA for execution. Software waits until it receives an acknowledgement from the FPGA indicating that all operations are complete. Finally, it issues the request to fetch the output table from DRAM on the FPGA and display it on the screen.

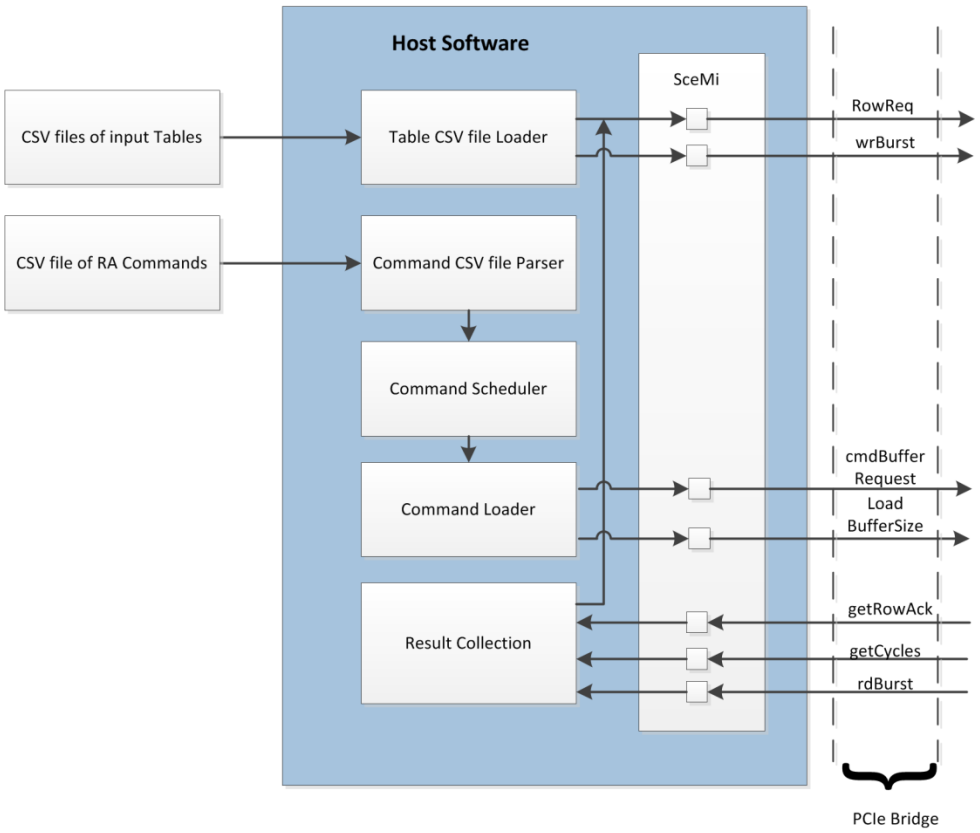

Figure 4: Host software block diagram

4.2 Row Marshaller & Mux

The Row Marshaller performs address translation and manages data bursts to provide an easy interface for the RA operators to access tables in DRAM (Figure 5). Given the table specifications (number of rows and columns), it is capable of reading a specific row of a table or reading an entire table. Because the DDR controller provides 256-bit bursts of data, the marshaller has to aggregate, split or truncate DDR bursts such that we can read table rows that are unaligned to 256-bit boundaries. In DRAM, tables begin at addresses aligned to 256-bits. Rows are packed as tightly as possible contiguously. All tables end with an end of table marker.

In addition, the Row Marshaller multiplexes requests and data from all the operator blocks onto a single memory controller. To reduce the size of the multiplexer, table data are sent out in 32-bit bursts to the operator blocks.

The Row Marshaller is also capable of handling one read request and one write request simultaneously. It is imperative that write requests to DRAM are prioritized over read bursts to avoid deadlocks in the system. Rows that have been processed must be written out before new rows can be processed by an operator. Another potential deadlock scenario can arise if the memory controller request FIFO is flooded with read requests, but the data pathway cannot drain because the write request to the memory cannot be enqueued. To resolve this, we used a large output buffer, and only issue read requests to the memory if there is enough space in the buffer to hold the output of that read request.

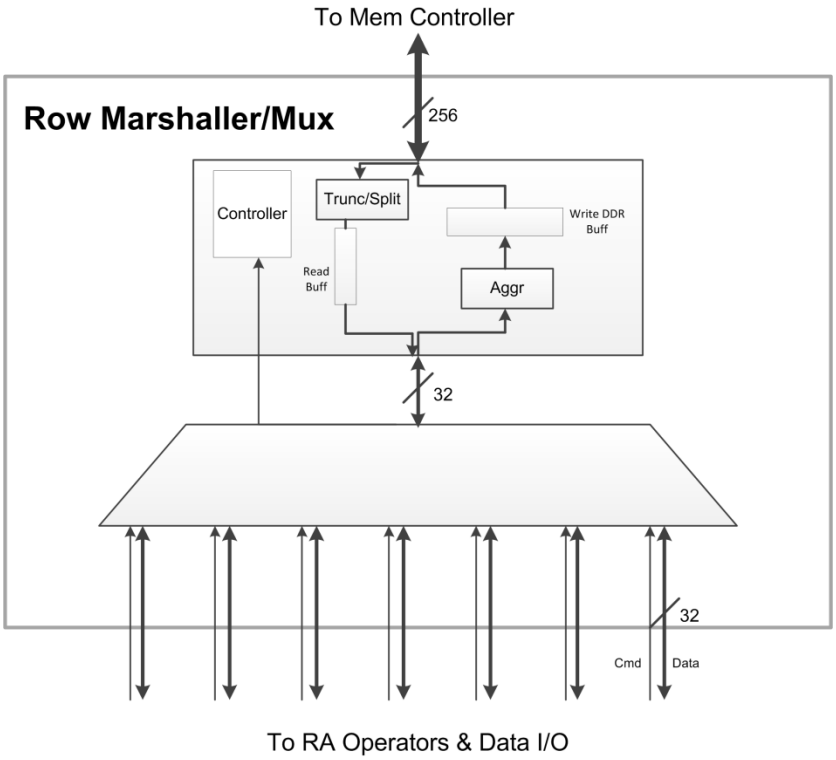

Figure 5: Row marshaller architecture

4.3 RA Controller

The RA Controller block receives RA commands from the host via SceMi and buffers all the commands in a BRAM. The controller runs a simple FSM that forwards a command to the appropriate RA operator block, waits for an acknowledgement from that operator and issues the next command. Upon completing all the commands, the controller acknowledges the software via SceMi.

The RA controller may send commands simultaneously to several RA operator modules such that the operators may stream rows among each other without storing intermediate tables to memory. The decision whether to do this will be made in software and encoded in the commands.

Commands are packed as structs and encoded as enums in both software and Bluespec. Each command contains all the information needed by the operator including table metadata:

```
struct CmdEntry {
   CmdOp op;
   uint32_t table0Addr;
   uint32_t table0numRows;
   uint32_t table0numCols;
   uint32_t outputAddr; //Addr for output table
   RABlock outputDest; //Store back to memory or pass to another RA operator
   RABlock inputSrc; //Where to get the rows, from mem or another RA operator
   //Select
   uint32_t numClauses;
   SelClause clauses[MAX_CLAUSES];
   ClauseCon con[MAX_CLAUSES-1]; //AND/OR connectors between clauses
   //Project
   uint32_t colProjectMask;
   //Union/Diff/Xprod
   uint32_t table1Addr;
   uint32_t table1numRows;
   uint32_t table1numCols; };
```

4.4 RA Operators

All RA Operators have similar interfaces:

- a command/ack interface to the RA controller
- an address/data interface to the Row Marshaller
- data interfaces to the other RA operators

Operations on the tables are typically done on a row by row basis. Rows are obtained from either the Row Marshaller (i.e. memory) or from other RA operator modules and buffered within the operator module. The operation is performed and the new sets of rows are sent back to the Row Marshaller or to other RA modules for further processing.

4.4.1 Selection

The Selection module filters rows of a table based on a set of predicates joined by AND or OR (Figure 6). Predicates are comparisons between an attribute and a value (e.g. age > 10) or two attribute (income > expense). We support up to 16 predicates in disjunctive normal form with a maximum of 4 AND's and 4 OR's. Each predicate is evaluated by a comparator block, which outputs a qualify signal to indicate true or false. These qualify signals are mapped to the input of a binary tree of AND/OR gates. The output of the tree indicates whether the row should be discarded or kept.

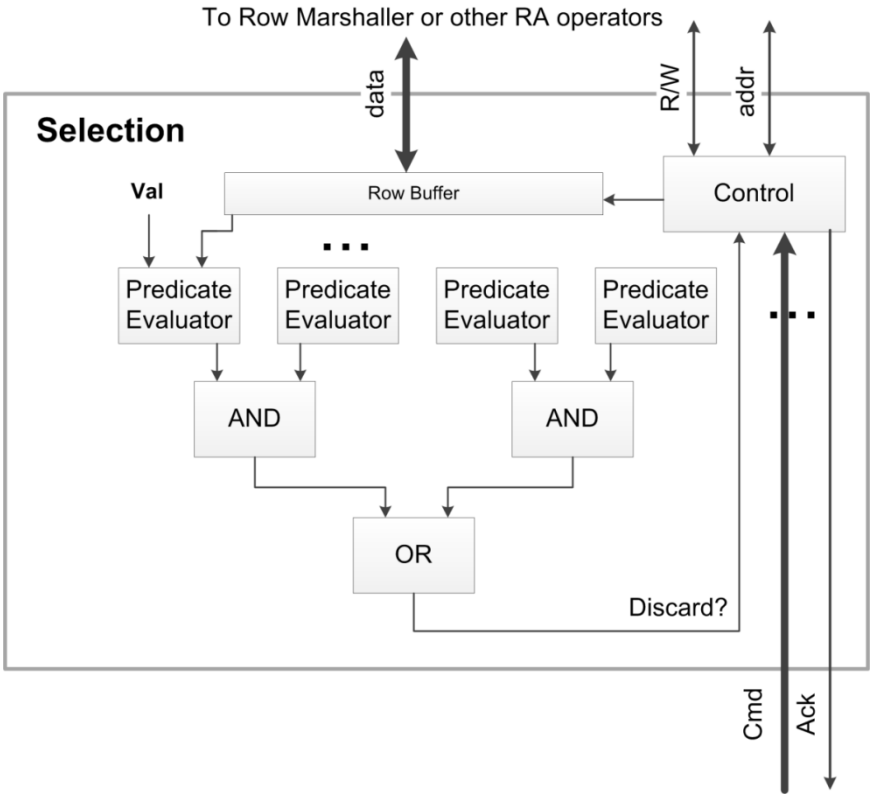

Figure 6: Select operator architecture

4.4.2 Projection

The Projection module filters columns of a table encoded in a column name mask (Figure 7). The column name mask uses one-hot encoding, where each bit corresponds to a column name. If the corresponding bit for a column name is 1, the column data field will be put into the output buffer. Otherwise, it is discarded. Projection operates directly on the incoming 32-bit bursts (which conveniently corresponds to the data width of a column), and do not have to buffer the entire row.

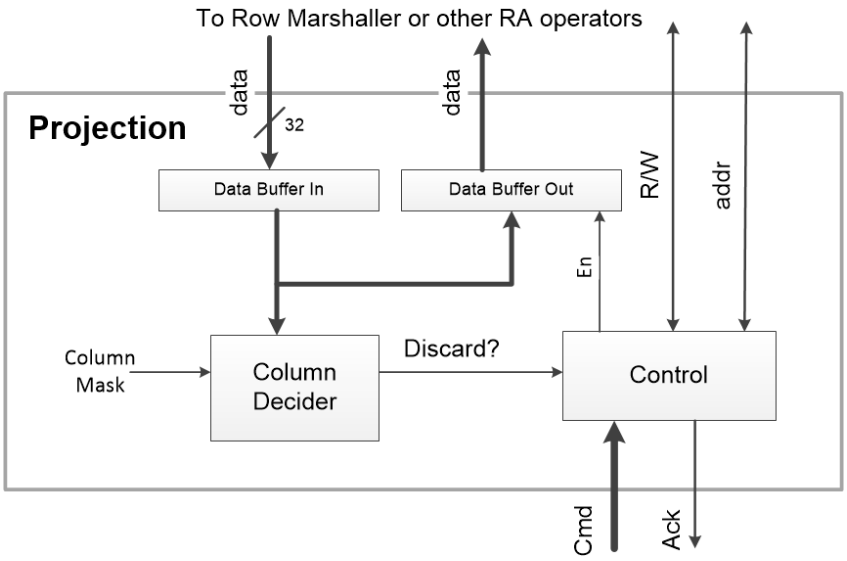

Figure 7: Project operator architecture

4.4.3 Cartesian Product

The Cartesian Product Module combines two input tables of unique set of table column names using nested loops. The module will stream in the rows of first table, and for each row from first table, it loops through second input table, and concatenate the rows. Concatenated rows are streamed either back to Row Marshaller or forward to other RA operators.

4.5.4 Union

The Union operator outputs the rows that are either in table 1 or table 2 (Figure 8). This is implemented by sequentially streaming in all rows of the first table followed by the second table. The Union operator requires that duplicates are removed, and this is achieved by comparing the rows of table 1 with table 2 using a nested loop.

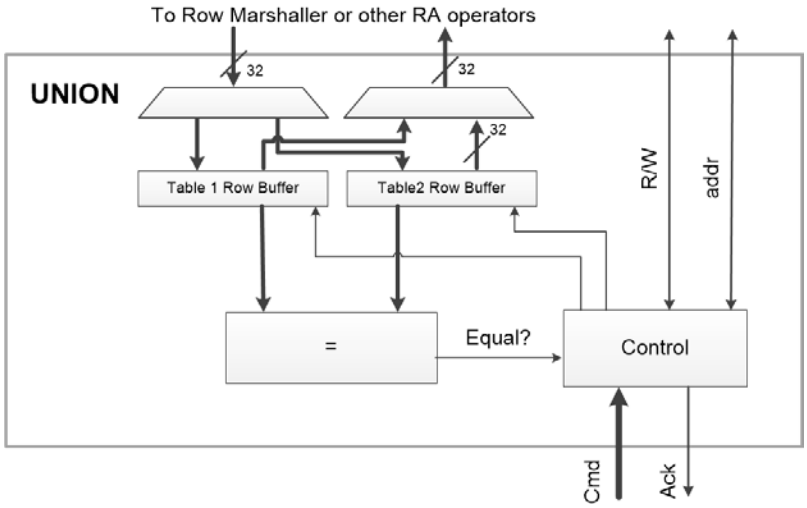

Figure 8: Binary operator architecture

4.4.5 Difference

The Difference operator selects rows that are in the first table but not in the second table (note that order matters). This is implemented using nested loops, where rows of the first table are searched for in the second table. If no match is found, the row is outputted.

4.4.6 Deduplication

Relational algebra semantics require duplicates to be removed after a Projection operation. However, duplication removal is an expensive operation on an unsorted dataset. Thus we decided to create a separate module for this purpose to give the flexibility of removing duplicates at any point in the series of commands (instead of always after Projection). The duplicate elimination block accepts an input table and generates an output table where each row is unique again using nested loops.

4.5 Inter-operator Data Bypassing

A key optimization of the system is the ability to pass rows of data between the operators without intermediate storage to memory. This drastically saves memory bandwidth and execution time. A relational algebra query can be thought of as a dependency/dataflow graph (directed), where each operator is a node and data stream is an edge. In our system, a node can have either one (unary operator) or two (binary operator) inputs, and any number of the same outputs. The dependency tree always ends with a single output table. We enabled a chain of operators concurrently if:

1) The operators (except for the first one) forms a singly link of unary operators
2) Each operator has a single target output
3) No structural hazards of operators

Figure 9 illustrate an optimized relational algebra query graph representation, where the blocks of same color are executed simultaneously.

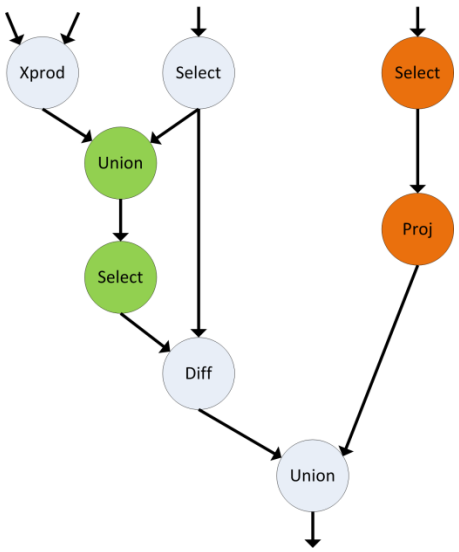

Figure 9: Directed Graph representing a relational algebra query.
The color filled operators are simultaneously enabled

Finding operators to enable simultaneously is done in software by the command scheduler. The scheduler sequentially searches the commands to identify dependencies and reorders the command buffer such that operators that can be enabled together are next to each other in the buffer. The command struct encodes the source and destination of the data stream such that the operators know where to fetch and output the data. On the FPGA, the RA Controller looks at the command buffer, and keeps issuing commands until the destination of the command is memory. It then waits for an acknowledgment from all of the simultaneously enabled blocks before issuing the next set of commands.

On the hardware side, each operator has multiple 32-bit wide input and output FIFOs each connected to a different destination or source. Depending on the data source and destination encoded in the command, the operators will get or put data from or into the corresponding FIFO. Figure 10 shows the exact connections at the top level.

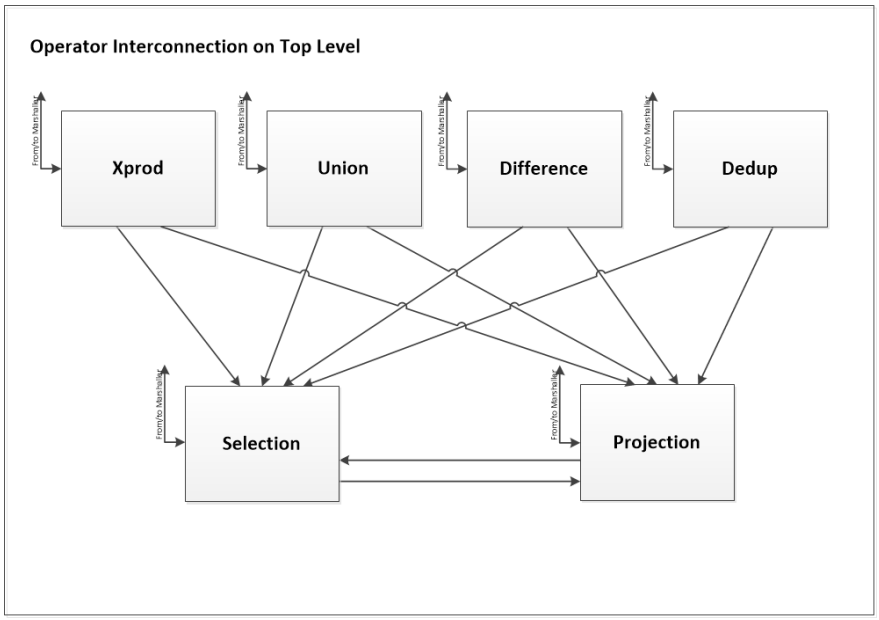

Figure 10: Inter-operator connections

5. Implementation Evaluation

We implemented the RA processor in Bluespec compiled using Xilinx ISE 13.4 for the XUPV5 development board. A breakdown of the resource utilization and number of lines of code in our design is presented below (Table 1, Table 2) . The system is compiled to operate at 50MHz, but can reach a max operating frequency of **55.79MHz** according to the synthesis report. We used the existing PCIe and DDR memory controller modules provided in class or by the TA.

Table 1: FPGA resource utilization

Modules	Slice Registers	LUTs	BRAM
Total(RAProcessor + SceMi + DDR)	34,649 (50%)	59,328 (85%)	71 (47%)
RowMashaller	2804	6627	0
Controller	4570	6277	29
Selection	3137	19633	0
Projection	739	654	0
Xprod	1935	1478	0
Union	1939	1983	0
Difference	1875	1949	0
Dedup	1822	1970	0

Table 2: Lines of BSV code (not including test benches or SceMi)

Bluespec Modules	Lines of Codes
procTop.bsv	90
RowMashaller.bsv	549
Controller.bsv	321
Selection.bsv	263
Projection.bsv	191
Xprod.bsv	279
Union.bsv	357
Differerence.bsv	260
Dedup.bsv	282

We did not encounter significant timing or area problems compiling for the FPGA. However, our design is using the majority of the LUTs on the FPGA. Most of the area is used by the Selection module and Row Marshaller. This primarily due to large muxes and predicate evaluation blocks. The critical path of the system is limited by the muxes in the Row Marshaller.

We did encounter some random data corruption of DDR data when moving from simulation to the FPGA. Occasionally, DDR bursts would contain unexpected bits. The problem disappeared after recompiling, but the root of the problem is still unknown. We had thought the problem was caused by the ResetXactor in SceMi possibly initializing the controller to a bad state, but removing the ResetXactor, we still saw data

corruption in some compiles. We guess that the problem could be caused by DDR timing constraints not being set properly.

5.1 Performance Test Setup

Performance of the RA processor was compared against SQLite. Each RA operator was translated to its equivalent SQL query and executed in both hardware on the RA processor and in software using SQLite. Table 3 summarizes the queries that were tested.

In hardware, the execution time was measured by using performance counters to record the number of cycles from the start of execution of the operator to the acknowledgement received from the operator. The cycle count divided by the frequency (50MHz) gives the execution time. Note that SceMi/PCIe is not considered in the benchmarks since it is a bottleneck created by Bluespec.

In software, internal SQLite timers were enabled to report the execution time of each query. In an attempt to equalize with the hardware platform, the following steps were taken:

- **SQLite database was placed in RAMDisk** since on the FPGA, the database resides in memory
- **SQLite outputs were suppressed** by counting the output number of lines using "select count(*)". This is because we are only concerned with computation time and not the overhead of stdout or its redirection.

SQLite was running on a Thinkpad T430 configured with Core i7-3520M @ 2.90Ghz, 1x8GB DDR3-1600.

Table 3: Queries tested

Table Configuration	Relational Algebra Query	SQL Query

1 table
100,000 x 30
	`SELECT,starLong,tableOut,`
	`mass,>,80000,`
	`AND,pos_x,>,10,`
	`OR,pos_x,<,pos_z,`
	`OR,col12,>,col14,`
	`AND,col20,<,col21`
		`select count(*) from (`
		`select *`
		`from starLong`
		`where mass > 80000`
		`and pos_x > 10`
		`or pos_x < pos_z`
		`or col12 > col14`
		`and col20 < col21);`

1 table
100,000 x 30
	`PROJECT,starLong,tableOut,`
	`pos_x,col19,col25,col29`
		`select count(*) from (`
		`select pos_x,col19, col25, col29`
		`from starLong);`

2 tables
1000 x 30
	`UNION,starMed1,starMed2,starUnion`	`select count(*) from (`
		`select * from starMed1`
		`union`
		`select * from starMed2) ;`

2 tables
1000 x 30
	`DIFFERENCE,starMed1,starMed2,starDiff`	`select count(*) from (`
		`select * from starLong`
		`except`
		`select * from starLong);`

2 tables
1000 x 30
	`XPROD,starMed1,starMed2,starXprod`	`select count(*) from (`
		`select * from starMed1, starMed2`
		`);`

1 table 1000 x 30	`DEDUP,starMed1,starOut`	`N/A`

2 tables 1000 x 30	XPROD,starMed1,starMed2,starXprod RENAME,starXprod,0,iOrder0,1, mass0,8,phi0 SELECT,starXprod,starFiltered, iOrder0,=,iOrder, AND,phi0,>,1, AND,mass0,>,mass PROJECT,starFiltered,starOut,mass0	select count(*) from (select s1.mass from starMed1 s1,starMed2 s2 where s1.vx > s2.vx and s1.phi > 1 and s1.mass > s2.mass);

5.2 Results

Figure 11 summarizes the performance results. Overall, the FPGA RA processor performs on par with SQLite in Select (filtering) operations, but is an order of magnitude slower in all other operations. This is in line with our expectations. Select is computationally more intensive than all other operators, which benefits from the parallel and dedicated predicate evaluators on the FPGA. On the other hand, other operators are memory bandwidth bound, and our nested loop implementations of Union, Difference, XProduct and Deduplication exacerbate the problem. A major bottleneck of our architecture is the data burst width between blocks and the RowMarshaller, which is set at 32-bit wide. This means that at full speed, our design only streams data out of DRAM at **200MB/s** (32 bits @ 50MHz) whereas the max transfer rate of DDR2-400 RAM on the XUPV5 is 3200MB/s [2], disregarding memory controller overheads. The 32-bit width was initially chosen because of resource limitations and timing concerns since there are many operators multiplexed onto the same DRAM controller and multiple FIFOs to stream data between the operators. However, presumably, we can increase the width of the bursts to 256-bits on a larger FPGA (as provided by the DRAM controller), which will increase the memory access bandwidth to **1600MB/s**. This could potentially give us an order of magnitude improvement in performance for all the operators.

In addition, the software is running on an x86 platform with a DDR3-1600 RAM module which has a peak transfer rate of 12.8GB/s [3]. Despite the OS memory management overheads, this is still a massive improvement over the DDR2-400 on the XUPV5. Caching and prefetching also add to the advantages.

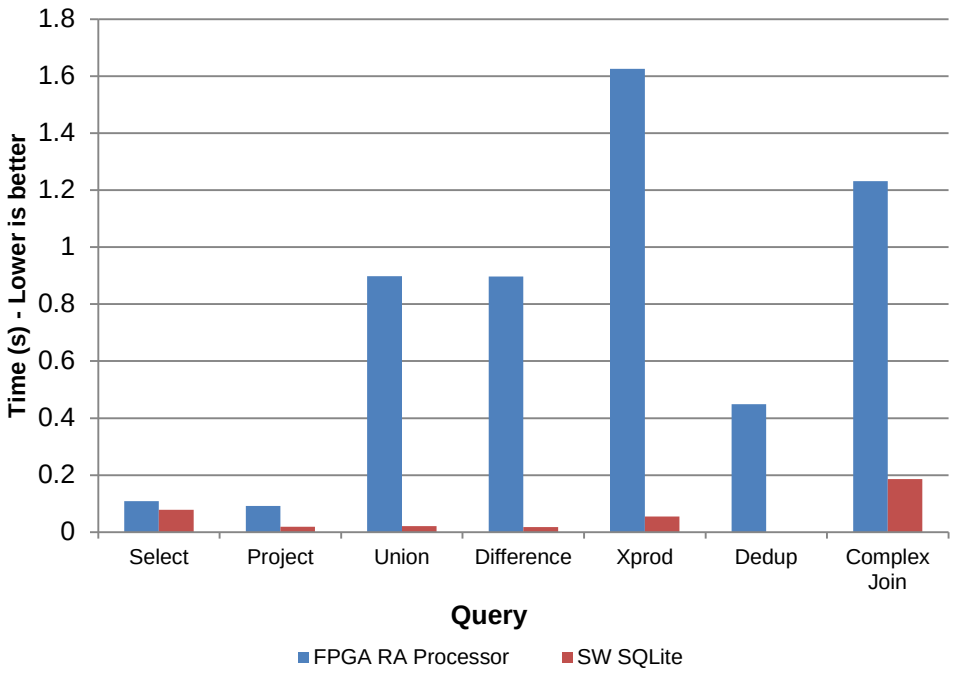

Figure 11: Query execution time comparisons

We evaluated the performance of Select in further detail since it seems to benefit from FPGA acceleration. We varied the number of predicates of the Select statement and measured the performance of each against SQLite (Figure 12). By design, the FPGA RA processor can support up to 16 predicates, and can evaluate all of these in parallel in one cycle. Thus we see that the execution time is relatively constant for up to 16 predicates. Note that the small bump at 4 predicates exists because there were fewer lines filtered out with that particular Select statement, and hence more memory write operations.

SQLite execution time increases linearly with the number of predicates due to increased computational complexity for each row. Extrapolating this graph, we can conclude that for large and complex Selection predicates, FPGA acceleration will result in notable speedups.

Finally, we verified that the inter-operator data bypass was giving us the expected speedup. The "complex join" query (Table 3, last row), was ran on the FPGA with bypass turned on and off. This query consists of a XProd followed by a Select and a Project, which can all be enabled at the same time for bypassing without intermediate storage to DRAM. We measured **3.414s with bypass disabled** versus **1.231s with bypass enabled**. This is a ~3x speedup. Of course, this speedup factor is data and query dependent, but overall, bypass does improve performance.

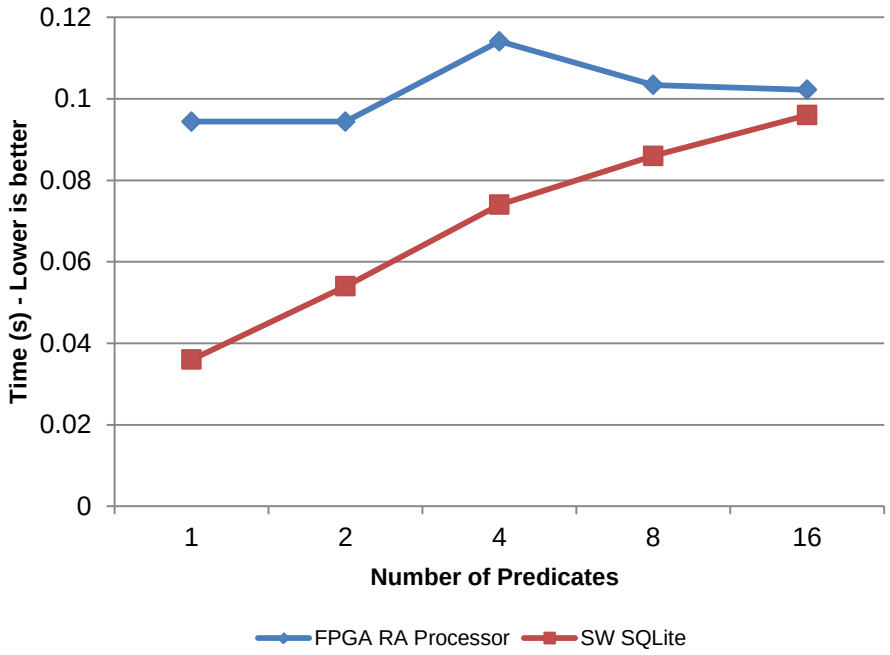

Figure 12: Select (Filter) execution time with varying number of predicates

6. Design Exploration

We analyzed the cycles required for each operator to explore possible optimizations of the design (Table 4). It is clear that most of the cycles are spent on streaming data to/from DRAM or other operator blocks. The binary operators must stream in/out two tables and may suffer additional latency penalties because their outer loop breaks sequentially access of DRAM.

Table 4: Cycle analysis

Operator	Cycle Analysis
Selection	NUM_COLS cycles to buffer each row; 1 cycles to evaluate each row based on predicates NUM_COLS cycles to write back the row if not filtered
Projection	1 cycle to process each column of each row and write back if not filtered
Union	Outer loop: NUM_COLS_A cycles to buffer tableA's row; NUM_COLS_A cycles to write back tableA's row Inner loop: NUM_COLS_B cycles to buffer tableB's row 1 cycle to compare row match (duplicate check) NUM_COLS_B cycles to write back tableB's row if no duplicates
Difference	Outer loop: NUM_COLS_A cycles to buffer tableA's row; Inner loop: NUM_COLS_B cycles to buffer tableB's row 1 cycle to compare row match (duplicate check) NUM_COLS_A cycles to write back tableA's row if no duplicates
XProduct	Outer loop: NUM_COLS_A cycles to buffer tableA's row; Inner loop: NUM_COLS_B cycles to buffer tableB's row NUM_COLS_A + NUM_COLS_B cycles to write back concatenated rows
Deduplication	Outer loop: NUM_COLS_A cycles to buffer tableA's row; Inner loop: NUM_COLS_A cycles to buffer tableA's row 1 cycle to compare row match (duplicate check) NUM_COLS_A cycles to write back tableA's row if no duplicates

Exploration point: Increasing data burst width

As mentioned in Section 5, using 32-bit data bursts is a key limitation of the system. Increasing the width of the bursts to up to 256-bit will reduce the number of cycles required to buffer a row and potentially result in orders of magnitude of improvement in performance. The tradeoffs are higher resource utilization and lower frequency, which could be alleviated by using a larger FPGA or pipelining.

Exploration point: Maximizing memory bandwidth

To always keep the memory busy, we could add additional row buffers to each operator such that while it is evaluating one row, it can simultaneously start buffering the next row. This can better hide the processing latency of each row and ensure that operator is constantly obtaining data from memory.

Exploration point: Larger, faster DRAM; Higher clock speed

Using DDR3 or faster DDR2 modules along with a higher clock speed (>50MHz), could alleviate the bandwidth limitations. Higher clock speeds can be attained by pipelining the multiplexers in the Row Marshaller to break up the critical path. As well, given larger DRAM modules we can run larger datasets to generate more accurate benchmark results.

7. Conclusion and Future Work

We successfully implemented a standalone relational algebra processor on the FPGA. Based on our evaluation, we have found that data intensive operators are not beneficial for execution on the FPGA, but given sufficient bandwidth and complex queries, filtering operators can outperform conventional database software running on an x86 PC.

A future opportunity for the project is to integrate such a processor in a hard drive or SSD controller to directly operate on the stored data. Furthermore, we can integrate the RA processor with SQLite as an accelerator. After generating a query plan, SQLite can decide based on the query complexity and type whether to offload the computation to process on the FPGA controller.

8. Acknowledgements

We would like to thank Prof. Arvind, Richard Uhler and Sangwoo Jun for mentoring our project and providing suggestions for improving our design!

9. References

[1] *Jun Yang*, RA: A Relational Algebra Interpreter, `http://www.cs.duke.edu/~junyang/ra/`, Accessed May, 2013

[2] *Xilinx*, XUPV5-LX110T User Manual, `http://www.xilinx.com/univ/xupv5-lx110T-manual.htm`, Accessed May, 2013

[3] *Crucial*, DDR3 Memory Speeds, `http://www.crucial.com/support/memory_speeds.aspx`, Accessed May, 2013