A Virtual Machine Model for Accelerating Relational Database Joins using a General Purpose GPU

Kevin Angstadt
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
kaa2nx@virginia.edu

Ed Harcourt
Department of Computer Science
St. Lawrence University
Canton, NY 13617
edharcourt@stlawu.edu

ABSTRACT

We demonstrate a speedup for database joins using a general purpose graphics processing unit (GPGPU). The technique is novel in that it operates on an SQL virtual machine model developed using CUDA. The implementation compiles an SQL statement to instructions of the virtual machine that are then executed in parallel on the GPU. We use the three-dimensional structure of the CUDA grid and thread model to perform a join on up to three relations at a time. Query execution results in speedups of 2 to 60 times on consumer-level GPUs depending on the size of the result set.

Author Keywords
GPGPU, SQL, virtual machine, join, relational database

ACM Classification Keywords
D.1.3 Concurrent Programming: Parallel Programming; H.2.4 Database Management: Parallel Databases

INTRODUCTION

A GPU, the major computational resource on a graphics card, has a primary role of computing and rendering images on a computer monitor. The massively parallel architecture of these chips has also been harnessed by researchers and applied to many computationally intense problems. Because GPUs are found in almost all modern computers, moving data processing to a host computer's GPU is a cost-effective method for decreasing execution time. While modern, consumer multi-core CPUs are designed with four to twelve simultaneously-executing threads, current, consumer-level GPUs can execute thousands of threads simultaneously. Such computational power is created at the expense of independent thread execution and large caches; however, the GPU's SIMD architecture is ideal for data processing that executes the same, independent calculation over a very large data set.

GPU manufacturers, such as AMD and NVIDIA have released APIs for their hardware, and the OpenCL framework also allows for the programming of GPUs. NVIDIA's API and framework, together known as CUDA [11] (Compute Unified Device Architecture), provides extensions to C/C++ as well as a programming interface for utilizing the massively parallel GPU. CUDA allows the programmer to control the memory spaces of both the host device (CPU) and the GPU, inter-thread communication, and mapping of threads and thread blocks to GPU hardware.

Relational database queries often fit a SIMD execution pattern; for example, the operations in a *where*-clause are mapped to every row in a table. A database *join* occurs when two or more source tables are combined into one resulting table based upon pre-defined and static relationships between rows and columns in the tables. The Structured Query Language (SQL), used by most relational database management systems (RDBMS), defines several different types of joins, but the most generic is the *cross-join*. Other joins can be defined in terms of the cross-join. Mathematically, the cross-join of two or more tables computes the Cartesian product (or cross-product) of the rows from each table (*e.g.,* with two tables all possible pairs of rows, three tables all possible triples of rows, etc.). The entire cross product is rarely meaningful. Predicates on the rows of the resulting cross product then restrict the resulting rows to those of interest. Such computations are simple to represent in SQL but are extremely compute-intense.

The scope of this research is to extend a GPU-based SQL virtual machine to allow for the execution of SQL statements containing joins and to demonstrate the efficacy of such execution methods. While most GPU-based database implementations utilize various parallel primitives, our implementation instead executes queries on an SQL virtual machine. The virtual machine executes on either the host CPU or a CUDA capable GPU allowing for performance comparisons.

RELATED WORK

Using GPUs to accelerate computations is well-established [10] and has a history of being applied to database operations [4, 6, 7]. Our work differs in that it uses an SQL virtual machine to represent the parallelism in the cross join and to map it directly to the geometry of the GPU.

This paper builds directly on the work presented in [1, 2, 3] by adding the join operation, which was not part of the original framework, to the Virginian database. In its initial form, this database was built directly on the SQLite Virtual Machine [12]; however, the current virtual machine implementation,

HPC 2015, April 12-15, 2015, Alexandria, VA, USA
© 2015 Society for Modeling & Simulation International (SCS)

though inspired by SQLite, is completely custom. The virtual machine represents a compilation target for SQL that can then be executed on either a host CPU or a GPU. The authors show significant speedup in non-join related SQL queries. The Virginian project introduced two new SQL virtual machine instructions for parallel processing, the `Parallel` and `Converge` instructions, to denote the instructions to execute on the GPU. We extend these instructions further for our GPU implementation to allow for parallel joins.

The work [5, 9, 8] includes join queries on GPUs by relying on a set of parallel primitives and produce speedups of 2 to 27 times. A primitive is a function implemented directly as an independent CUDA kernel such as *sort*, *map*, and *filter*. The authors then implement and evaluate various algorithms for joining tables in terms of these primitives including nested loop joins, sort-merge joins, and hash joins. Our work differs in that we implement an SQL virtual machine rather than individual primitives. SQL statements are compiled directly to a virtual machine opcode model rather than represented in terms of higher level CUDA kernel primitives. The benefits of executing a virtual machine as a kernel rather than parallel primitives is thoroughly explored in [2].

IMPLEMENTATION

Consider tables $\mathbf{T}_1$ and $\mathbf{T}_2$ that share an attribute $\mathbf{c}_2$. The SQL query below computes a cross-join with a predicate that yields the rows in the cross product where the values of $\mathbf{c}_2$ in each table are equal. This SQL statement is the equivalent of the *natural*-join.

```
SELECT * FROM T1,T2 WHERE T1.c2 = T2.c2
```

For example, the cross join of tables $\mathbf{T}_1$ and $\mathbf{T}_2$ in Figure 1 with the predicate $\mathbf{T}_1.\mathbf{c}_2 = \mathbf{T}_2.\mathbf{c}_2$ (denoted $\mathbf{T}_1 \bowtie \mathbf{T}_2$) contains nine rows but the predicate restricts the result table to just the three highlighted rows.

A join such as this can be implemented through *nested loops*, where each loop iterates over an input table and emits rows matching the predicate. Such an implementation is used by SQLite [12]. We adopt this algorithm for our virtual machine as well and use the three-dimensional CUDA thread topology to implement the loop nesting directly.

SQL to Opcode Translation

The implementation parses an SQL query and generates an *abstract syntax tree* (AST) for the query. The AST is then processed in several passes to generate a virtual machine program that represents the query.

Consider the following SQL join query, (part of our benchmarking queries listed in the appendix):

```
SELECT test.id, test1.uniformi,
    test.normali5 FROM
    test,test1 WHERE
    test1.uniformi > 60 AND
    test.normali5 < 0
```

Here, the two tables `test` and `test1` each have six columns: the first three are integer columns and the last three are floating-point. The query above computes the cross product of the two tables restricting the result table so that the column `test1.uniformi` is greater than 60 and the column `test.normali5` is negative. The resulting virtual machine program is shown in Figure 2.

$\mathbf{T}_1$

c_1	c_2
1	w
2	z
3	z

$\bowtie$

$\mathbf{T}_2$

c_2	c_3
x	5
z	6
w	7

=

$\mathbf{T}_1 \bowtie \mathbf{T}_2$

c_1	c_2	c_2	c_3
1	w	x	5
1	w	z	6
1	w	w	7
2	z	x	5
2	z	z	6
2	z	w	7
3	z	x	5
3	z	z	6
3	z	w	7

Figure 1: The cross join, $\mathbf{T}_1 \bowtie \mathbf{T}_2$. The rows in the *natural join* are highlighted.

```
 0: Table          0  0  0        0
 1: Table          1  0  1        0
 2: ResultColumn   0  0  0       id
 3: ResultColumn   0  0  0 uniformi
 4: ResultColumn   0  0  0 normali5
 5: Parallel       0  0 16        0
 6: Column         3  0  1        0
 7: Integer        0 60  0        0
 8: Le             3  0 14        0
 9: Column         4  1  0        0
10: Integer        1  0  0        0
11: Lt             4  1 13        1
12: Invalid        0  0  0        0
13: Rowid          2  0  0        0
14: Result         2  3  0        0
15: Converge       0  0  0        0
16: Finish         0  0  0        0
```

Figure 2: A sample virtual machine program generated by the SQL compiler.

The virtual machine instructions are explained in [1]; however, this instruction set only operates over a single source table. We therefore extend this to support multiple tables using *table cursors*, which are similar to cursors in SQLite. A table cursor is a positive integer that acts as a pointer to the table data associated with a specific instruction. A cursor is assigned to a table through the third parameter of `Table` instruction, which opens a handle to a table. Similarly, we extend the `Column` and `Rowid` instructions, used for accessing data in a table, to read from a specific table via its cursor. More formally, the syntax of these three modified opcodes is:

Table [table id]**,** []**,** [cursor]**,** []
Column [destination register]**,** [source column]**,** [cursor]**,** []
Rowid [destination register]**,** []**,** [cursor]**,** []

For the sample opcode program given in Figure 2 and its corresponding SQL statement given previously, lines `0` and `1` open cursors `0` and `1` to tables `test` and `test1`, respectively. Lines `2` through `5` configure the result table and invoke the parallel portion of the SQL query. Line `6` reads and copies column `0` of the row at cursor `1` to register `3`. This value is compared with `60` (loaded in line `7`) in line `8`. This constitutes the first portion of the `WHERE` clause. A similar comparison is then made in lines `9` through `11` for the second portion of the clause. The value of this column is stored in register `4`. If the row is still valid after this filtering, we load the primary key for the row at cursor `0` in line `13`. We then copy three registers, beginning with register `2` to the result table in line `14`. Lines `15` and `16` complete and clean up the query.

These extensions to the syntax of the instructions maintain backwards compatibility with the previous version of the Virginian database because cursors are passed using previously unused parameters in the instructions.

The execution of the virtual machine program is done at two levels. Instructions outside of the `Parallel` and `Converge` boundary initialize the query and are executed on the host CPU. The parallel portion (those between `Parallel` and `Converge`) of the query runs in a separate virtual machine, which can be implemented on the CPU or also on an accelerator (in our case, a CUDA kernel running on the GPU). The inherent looping over the source table data is obfuscated in this section of the opcode program. This is because the individual result rows in a join are independent. Instead of programming the iteration into the virtual machine program, we describe the processing necessary for a *single* data point in the join and allow the virtual machine (implemented as a kernel) to map this efficiently over all data points in the query.

Therefore, the code generated between the `Parallel` (line 5) and `Converge` (line 15) instructions and how that region can be mapped to the three-dimensional CUDA thread topology described later is of particular interest to us. This mapping essentially flattens the nested loop structure used to compute cross products and allows for all data points to be computed in parallel on the GPU, which we describe shortly.

Tablet Management

We present a brief overview of table data management; for a detailed description, see [2]. Database tables are typically stored using a balanced tree data structure (*e.g.,* a BTree). To take advantage of the grid topology of a CUDA-based GPU we instead store subsets of a table known as *tablets*, which partition the overall table vertically. Data within tablets is stored in column-major order, which allows for better data coalescing on GPUs and caching on CPUs. Each table also contains meta-data about its contents as well as space for keys, fixed-width, and variable-width data. To represent an entire table, tablets are organized into a linked list, and the SQL virtual machine processes one tablet at a time. Data in a tablet can be accessed in constant time via a pointer to the start of a column and a row offset. Currently we assume that source tables fit within one tablet (and the join result may span many tablets); though, it is straightforward to support larger source tables that span multiple tablets.

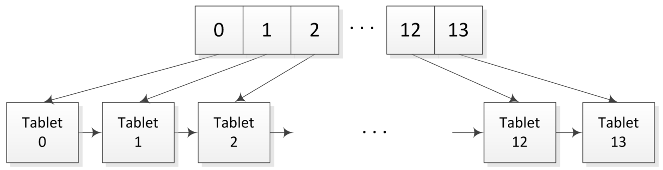


Figure 3: The result table is a linked list of tablets. To allow for constant time access to any row in the table, we maintain an array of pointers to each tablet.

When executing a query to join two tables, careful attention must be paid to the resulting table and its tablet representation. The result of a query that involves computing a Cartesian product can consume large amounts of memory. Given a table with m rows and a table with n rows, the length of the table resulting from the cross product can contain as many as $m \cdot n$ rows. Consider the example of crossing two tables, each with 3 500 rows. The result table will contain at most $3\,500 \cdot 3\,500 = 12\,250\,000$ rows. Because there is a limited amount of memory on a GPU, it is possible that this result cannot fit into the available global memory. Consequently, we use *mapped memory* to store both the data and results tablets. Mapped memory is main system memory that has been pinned and mapped via the NVIDIA CUDA API to the graphics card's memory space. The memory is *pinned* (page-locked) in that it cannot be swapped out by the operating system, and is then guaranteed to be available when accessed from either the host code or the CUDA kernel. By using mapped memory, we can allow for larger joins. One drawback, however, is that memory accesses must now travel across the PCI bus, which is significantly slower than global memory accesses on the GPU. As noted by Bakkum, using mapped memory is still faster than the combined time needed to copy data to and from the GPU as well as executing the code [2].

Suppose that a tablet can hold at most 1 000 000 rows (this number is dependent on the data in the table, but a tablet is no larger than 8MB), and we execute a query that produces the Cartesian product of two tables with 3 500 rows each. The resulting table will span

$$\left\lceil \frac{3\,500 \cdot 3\,500}{1\,000\,000} \right\rceil = \left\lceil \frac{12\,250\,000}{1\,000\,000} \right\rceil = 13 \text{ tablets.}$$

Prior to query execution, we do not know how many rows will be in the result, and so we conservatively allocate all 13 tablets. After query execution, we can delete any unused tablets before returning the result. Because tablets are stored as a linked list, the result table does not provide constant time access to its rows. To retain constant time access, we generate an array of pointers to each tablet in the result table. Instead of walking through the linked list of tablets, we can now directly access each tablet in constant time. Figure 3 shows an example result tablet structure that might be allocated for a query.

$\mathbf{T}_1$

c_1	c_2
1	w
2	z
3	z

$\bowtie$

$\mathbf{T}_2$

c_2	c_3
x	5
z	6
w	7

=

$\mathbf{T}_1 \bowtie \mathbf{T}_2$

(1,w,x,5)	(1,w,z,6)	(1,w,w,7)
(2,z,x,5)	(2,z,z,6)	(2,z,w,7)
(3,z,x,5)	(3,z,z,6)	(3,z,w,7)

Figure 4: The join of two tables can be constructed in a two-dimensional grid. Highlighted cells indicate rows that remain after filtering on the `WHERE` clause.

Query Execution

CUDA organizes threads into a grid of up to three dimensions. We exploit this 3D topological structure as it coincides nicely with the structure of a Cartesian product where a two table join is a two dimensional grid and a three table join is a three dimensional grid. Consider again the join of two tables T_1 and T_2 from the introduction.

```
SELECT * FROM T1,T2 WHERE T1.c2 = T2.c2
```

An alternative view of the result table is as a two-dimensional CUDA grid as in Figure 4. Here, the row index in the two-dimensional table corresponds to the row index of $\mathbf{T}_1$ while the column index corresponds to the row index of $\mathbf{T}_2$. We translate this directly to the dimensions of the instantiated CUDA kernel. Thread indices in the x-axis correspond to row indices of the first table in the SQL query, and thread indices in the y-axis correspond similarly to the second table in the query.

Each kernel thread has access to 1) the virtual machine program in constant GPU memory, 2) pointers to the data in the tables it requires as well as 3) pointers to the tablet structure to write back the result rows. For example, in Figure 4 each cell in the result table coincides with a thread and would contain a pointer to the appropriate rows in T_1 and T_2.

A high-level outline of the process of executing a query is:

1. Copy table meta-data to the GPU
2. Copy source tablets to the GPU
3. Allocate pointers to meta-data on the GPU
4. Launch kernel threads (described below)

A single thread operates on one item in the cross product interpreting the instructions between `Parallel` and `Converge` in the generated opcode program. Recall that these instructions are independent as they describe operations on a single data point in the cross product being evaluated in the query. Therefore, all kernels operate in parallel for all entries in the cross product of the source tables. All threads execute the same exact SQL virtual machine program on the table rows the thread has been assigned. The kernel implements a function for each virtual machine instruction. These functions run on the GPU only (`__device__` functions) and are called by the kernel threads. Much of the work prior to launching the kernels is in setting up the tablet structure for each thread. Each thread computes the source rows it is responsible for using the built-in CUDA variables `blockIdx`, `blockDim`, and `threadIdx`:

$$\texttt{row} = \texttt{blockIdx}.dim * \texttt{blockDim}.dim + \texttt{threadIdx}.dim$$

where $dim =$ `x` for the table at cursor `0` and $dim =$ `y` for the table at cursor `1`.

When writing rows to the result table, we must calculate the correct result row indices. Because a thread does not know which other threads will produce rows for the result tablets this requires synchronization between the kernel threads. For all threads within a block that have a valid result row, we *atomically add* 1 to a counter, `block`. Within the CUDA framework, an atomic add returns the current value of `block` to the thread, which we store as `place`. Within a block, each kernel now knows its result row offset. The threads are then synchronized to ensure that all threads have a `place` before proceeding. Next, for each block, we now atomically add `block` to another counter, and threads in each block share the returned value as `start_of_block`. This value indicates the offset in the result table of each block of threads for writing result rows. After another thread synchronization, we are guaranteed that each thread now has a valid `place` and `start_of_block`. The row index in a thread is therefore

$$\texttt{row\_index} = \texttt{start\_of\_block} + \texttt{place}$$

for all threads with a valid result row. With this index, writing to the result table does not require coordination between the threads. From the row index, each thread calculates the tablet and offset of the row relative to a given tablet:

$$\texttt{tablet\_index} = \left\lfloor \frac{\texttt{row\_index}}{\texttt{rows\_per\_tablet}} \right\rfloor$$

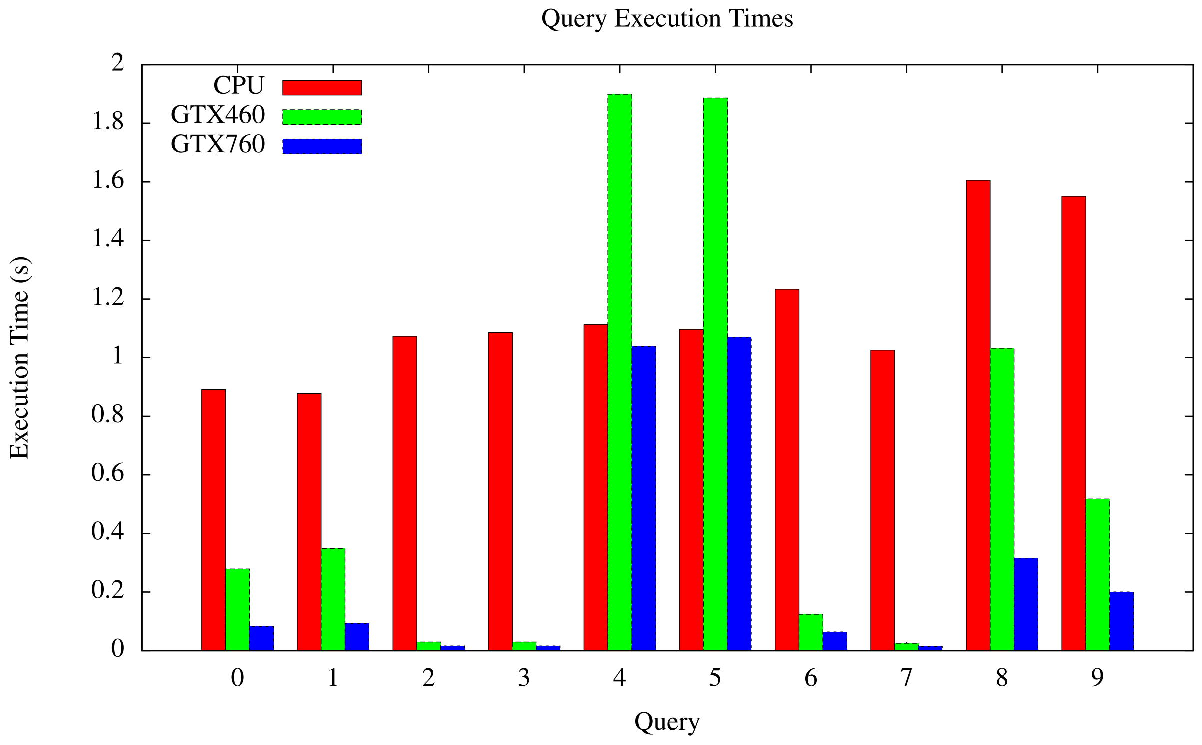
$$\texttt{offset} = \texttt{row\_index} - (\texttt{tablet\_index} \cdot \texttt{rows\_per\_tablet}).$$

Using theses values, the thread writes its row across the PCI bus to the result table stored in mapped memory.

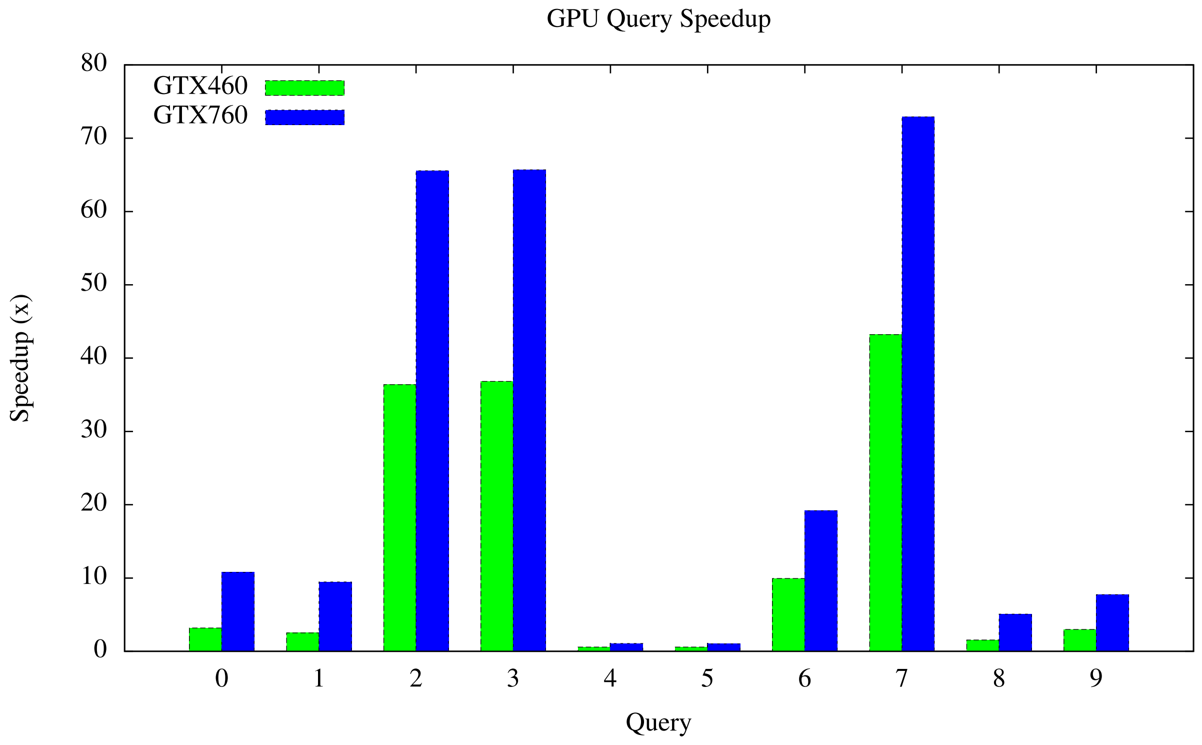
RESULTS

Tests were performed using two 3 500 row tables containing randomly generated values. We use the same source table layout as [2]. Each table consists of an integer primary key, and three columns of values each for both 32-bit integers and IEEE 754 32-bit floating point values. One column is randomly distributed across $[-100, 100]$, the second is a normal distribution with a sigma of 5, and the final column contains a normal distribution with a sigma of 20.

Tests were conducted both on an NVIDIA GTX460 GPU (336 CUDA cores and 1GB memory) and an NVIDIA GTX760 GPU (1152 CUDA cores and 2GB memory) using CUDA 6.5 and the NVIDIA 340.29 driver. The host CPU for



(a)



(b)

Figure 5: Queries show varied execution times (a) and speedup (b) on the GPU for different queries in the test suite.

	CPU	GTX460	GTX760
Integer	1.183	0.673	0.304
Floating Pt.	1.127	0.561	0.279
All	1.155	0.617	0.291

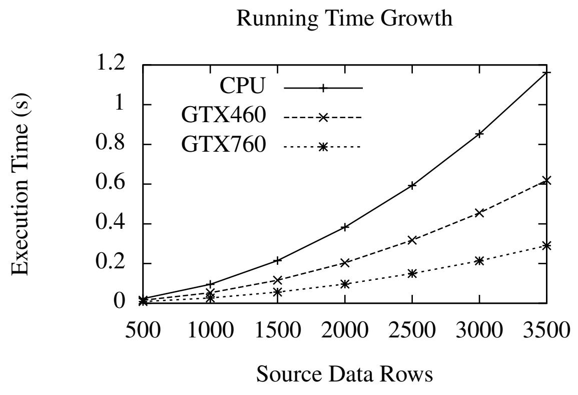
Table 1: Running times in seconds for CPU and GPU execution of integer and floating-point arithmetic queries.

both GPUs was a 2.6 GHz Intel Core i7 920 CPU running the 3.13.0-39-generic Linux kernel. Tests were executed ten times and the data presented here is the average of these runs.

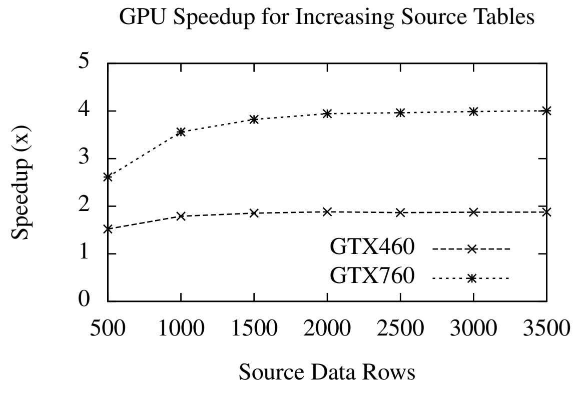
Figure 5 graphically demonstrates the differences in running times on both the CPU and GPU for ten different join queries. The mean execution time for all ten queries on the CPU was 1.155 seconds, and the mean GPU execution time was 0.617 seconds and 0.291 seconds for the GTX460 and GTX760 respectively. Even numbered queries contain predominately integer arithmetic, with each subsequent odd-numbered query executing the same query with floating-point data. Table 1 lists the average running times for each of these two categories on both the GPUs and the CPU. There is no significant difference in values between these integer and floating-point queries, which indicates that speedup is independent of the data type. Additionally, the GPU executed faster than the CPU on average for both tests.

Figure 6(a) depicts the average running time for our suite of ten queries for increasing source table sizes, and figure 6(b) represents this data as speedups. Performance on smaller table sizes is less due to memory writes making up a greater portion of the total execution time. Nevertheless, the GPU implementation of the SQL virtual machine executes approximately twice to four times as fast as the CPU virtual machine on average for this query suite.

Queries 4 and 5 in Figure 5 executed more slowly on the GTX460 than the CPU, and the GTX760 executed in approximately equal time compared with the CPU. We hypothesize that this is due to result table sizes. While all other queries output 2.25 million rows or fewer, these two queries output approximately 6 million rows each. The additional time required to write these results across the PCI bus to the host memory significantly slowed execution time. We then conducted additional benchmarking in order to verify that the memory writes are the limiting step during query execution. To test this hypothesis, we incrementally increase the number of result rows in a cross-join of two, 3 000 row tables. As represented by Figure 7, the GPU becomes less efficient until the GPU executes in the same time as the CPU. For the GTX460, this occurs at approximately 1.8 million rows, and at 4.5 million rows for the GTX760. These values will vary with the computation required by the query and the layout of the desired data in the source tables. The PCI bus was also a limiting factor in our tests because the test machine only supported PCIe2. The GTX760 is designed to utilize the additional throughput of PCIe3, and this additional throughput would push the break-even point even closer to a full Cartesian product.



(a)



(b)

Figure 6: Average performance on the ten query suite for increasing source table sizes. (a) measures running times and (b) measures speedup relative to CPU.

SQL joins that result in a massive number of result rows approaching a full Cartesian product are uncommon and are inherently problematic even in commercial RDBMSs. For more reasonable queries, in which the predicate filters most rows, the GPU remains extremely efficient for varying source table sizes. Figure 8 shows the growth of execution time as a function of source table size for a query with a restrictive predicate. This query limits the result table to be the same size as the input table by joining the two tables on their key, a common join operation in database queries. For this query, speedup ranges from 20 to 30 times on the GTX460 and 40 to 60 times on the GTX760.

CONCLUSIONS AND FUTURE WORK

This paper demonstrates the efficacy of accelerating database joins using an SQL virtual machine based GPU execution. Previous research has shown speedup for VM-based execution for single-table queries on GPUs, and our results indicate that this holds for multiple-table queries as well. Our implementation achieves equal or better speedups as compared with joins implemented with primitives-based kernels.

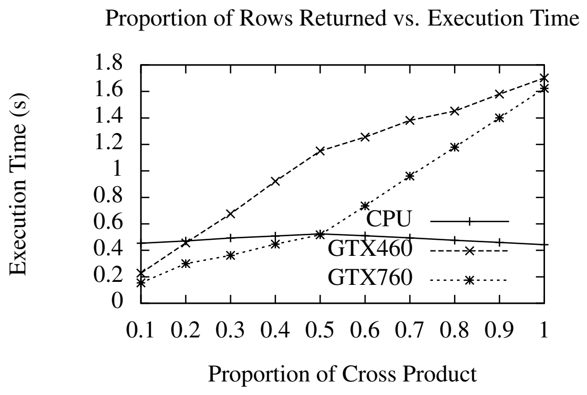


Figure 7: As fewer rows in the Cartesian product are filtered, the GPU becomes less efficient.

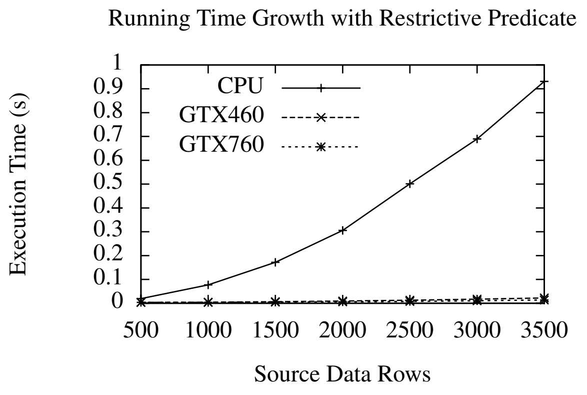


Figure 8: Average running times with restrictive predicate for increasing source table sizes.

Our extensions to the Virginian database system primarily demonstrate the techniques needed to allow for joining multiple tables. The next step would be to ensure that all stages of the virtual machine's execution are efficiently implemented. Although, we took care to implement a fair computational schema on both the CPU and GPU, kernel memory writes across the PCI bus are currently inefficient. The current method for copying results to the mapped memory space does not take full advantage of coalesced memory accesses, a feature which can be up to an order of magnitude faster than non-coalesced memory accesses [11]. Because memory writes are the limiting step in queries with joins, we anticipate such an implementation to improve the performance of the GPU virtual machine.

Although our framework does not currently support coalesced writes to mapped memory, our multi-dimensional kernel instantiation and use of mapped memory results in an average of 2x-4x speedup over CPU-based query execution. For joins with reasonably restrictive predicates, speedups can be as much as 20x-60x on consumer-level GPUs. Due to the relatively low cost of GPU hardware and its ubiquitous nature in modern computer systems, a framework such as this provides a low cost alternative to distributed RDBMSs for accelerating query processing.

Another interesting extension to this research would be the dual use of the CPU- and GPU-based virtual machines. In this scenario, pre-processing of the data could help to determine the most efficient virtual machine for query execution. In queries resulting in large amounts of data with relatively little computation, the CPU virtual machine would be selected; otherwise queries would be executed on the GPU. This would help avoid the cases where memory writes limit the overall performance of query execution, but still benefit from GPU speedup in the general case.

The current implementation technique is also limited to joining at most three tables at a time because of the three dimensional nature of CUDA thread blocks. Because joins in SQL are *closed* (the result of joining two tables is another table) in order to join more than three tables a query must be divided into multiple stages. By automating this process, our framework could then theoretically handle an arbitrary number of tables. Additionally, we would like to incorporate other join syntaxes into the SQL parser to allow for additional support of the SQL language. Such extensions allow for simple notation for several types of joins, including `inner`, `outer`, and `natural` joins.

More generally, a natural follow-up to this research would be a study on the scalability of this technique. Our tests use at most 3 500 rows in each source table. Do speedups remain consistent for larger source table sizes, or will tablet management and writes across the PCI bus subsume the overall computation time? Additionally, how does increasing the number of joined tables affect performance?

In addition to improving the efficiency of the software itself, another interesting research path would be multi-card implementations. Splitting data across multiple GPUs has a two-fold advantage: data can be processed more quickly and more data can be processed. Since all table data is stored in mapped memory rather than on the GPU itself, such an implementation could be straight-forward. Each graphics card would be responsible for a different section of the cross product, but each would access the same host memory space. Use of mapped memory also avoids the overhead of copying tables to multiple devices and joining the results back into a single result table after the kernel execution on separate devices completes.

The benchmarks associated with the Virginian framework are highly dependent on the hardware configuration of the test machine; using different hardware may demonstrate different speedups. Compared with more expensive higher performance graphics cards, our hardware was relatively inexpensive with lower performance.

REFERENCES

1. Bakkum, P. The Virginian Database. **`https://github.com/bakks/virginian/blob/master/README.md`**. Date accessed: February 12, 2015.

2. Bakkum, P., and Chakradhar, S. Efficient data management for GPU databases. Tech. rep., NEC Laboratories America, Princeton, NJ.
3. Bakkum, P., and Skadron, K. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, ACM (New York, NY, USA, 2010), 94–103.
4. Bandi, N., Sun, C., Agrawal, D., and El Abbadi, A. Hardware acceleration in commercial databases: A case study of spatial operations. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, VLDB Endowment (2004), 1021–1032.
5. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N. K., Luo, Q., and Sander, P. V. GPUQP: Query Co-processing Using Graphics Processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, ACM (New York, NY, USA, 2007), 1061–1063.
6. Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, ACM (New York, NY, USA, 2006), 325–336.
7. Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., and Manocha, D. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, ACM (New York, NY, USA, 2004), 215–226.
8. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst. 34*, 4 (Dec. 2009), 21:1–21:39.
9. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, ACM (New York, NY, USA, 2008), 511–524.
10. Kirk, D., and Hwu, W.-m. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann, 2012.
11. NVIDIA Corporation. NVIDIA CUDA programming guide. **http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf**, February 2014. Date accessed: February 12, 2015.
12. SQLite. **http://www.sqlite.org/vdbe.html**. Date accessed: February 12, 2015.

BENCHMARK QUERIES

Listed below are the queries used to evaluate the performance of our SQL virtual machine. These were adapted from [3] and [2].

```
0: SELECT test.id, test1.uniformi, test.
   normali5 FROM test,test1 WHERE test1.
   uniformi > 60 AND test.normali5 < 0
1: SELECT test.id, test1.uniformf, test.
   normalf5 FROM test,test1 WHERE test1.
   uniformf > 60.0 AND test.normalf5 <
   0.0
2: SELECT test.id, test.uniformi, test1.
   uniformi FROM test,test1 WHERE (test.
   id - test1.id) < 5 AND (test.id -
   test1.id) > -5 AND test.uniformi >
   test1.uniformi
3: SELECT test.id, test.uniformf, test1.
   uniformf FROM test,test1 WHERE (test.
   id - test1.id) < 5 AND (test.id -
   test1.id) > -5 AND test.uniformf >
   test1.uniformf
4: SELECT test.id, test1.uniformi, test.
   normali20 FROM test,test1 WHERE (
   test1.uniformi < test.normali20) AND
   (test.normali20 + 40) > (test1.
   uniformi - 10)
5: SELECT test.id, test1.uniformf, test.
   normalf20 FROM test,test1 WHERE (
   test1.uniformf < test.normalf20) AND
   (test.normalf20 + 40.0) > (test1.
   uniformf - 10.0)
6: SELECT test.id, test.normali5, test1.
   normali20 FROM test,test1 WHERE test.
   normali5 = test1.normali5 AND test.
   normali5 * test1.normali20 >= -5 AND
   test.normali5 * test1.normali20 <= 5
7: SELECT test.id, test.normalf5, test1.
   normalf20 FROM test,test1 WHERE test.
   normalf5 = test1.normalf5 AND test.
   normalf5 * test1.normalf20 >= -5.0
   AND test.normalf5 * test1.normalf20
   <= 5.0
8: SELECT test.id, test1.uniformi, test.
   normali5, test.normali20 FROM test,
   test1 WHERE test1.uniformi >= -1 AND
   test1.uniformi <= 1 or test.normali5
   >= -1 AND test.normali5 <= 1 OR test.
   normali20 >= -1 AND test.normali20 <=
    1
9: SELECT test.id, test1.uniformf, test.
   normalf5, test.normalf20 FROM test,
   test1 WHERE test1.uniformf >= -1.0
   AND test1.uniformf <= 1.0 or test.
   normalf5 >= -1.0 AND test.normalf5 <=
    1.0 OR test.normalf20 >= -1.0 AND
   test.normalf20 <= 1.0
```