Accelerating RDBMS Operations Using GPUs

Jason V Ma, Ryerson University

Available at: https://works.bepress.com/jason_ma/1/
Accelerating RDBMS Operations Using GPUs

Jason Ma *
Ryerson University
350 Victoria Street
Toronto, Canada
jason.ma@ryerson.ca

ABSTRACT
This survey paper reviews how the Graphical Processing Unit (GPU) can be leveraged by accelerating queries for the Relational Database Management System. Queries can be accelerated in many ways like optimizing primitives, using an opcode execution model, using an in memory database, kernel fusion, batching and data placement. Data transfer remains a significant bottleneck and thus the mitigation methods such as database compression, differential update and unified address space are reviewed.

Keywords
RDBMS, join, compression, database, GPU, CUDA, Nvidia, kernel fusion

1. INTRODUCTION
Graphical Processing Units (GPUs) are massively parallel processors which can be leveraged for High Performance Computing (HPC) due to their high performance with floating point numbers, matrix operations and computations that overlap with the graphics domain. The main obstacle preventing GPUs to be adapted for general computing were calculations had to be translated to the graphics domain so they can be represented as polygons and triangles via graphics APIs such as Microsoft’s DirectX or OpenGL.

GPUs were originally intended for the video game market to render complex 3D graphics because the CPU was unable to achieve the high frame rates needed to ensure a satisfying gaming experience. During the mid 1990s 3DFx Interactive (now defunct) introduced their Voodoo graphics card, the first mainstream GPU which revolutionized the gaming industry [28].

Around 2007 Nvidia and ATI (now AMD) introduced non-graphics APIs so general purpose (i.e. scientific, financial, engineering, etc...) calculations can be performed on their GPUs. AMD and Nvidia GPUs support the OpenCL [27] API which allows computations to execute on CPUs and GPUs. Nvidia also has their proprietary CUDA (Compute Unified Device Architecture) APIs which are usually the first to implement any new features.

Recently, researchers have attempted to accelerate database operations using GPUs. This paper reviews the results reported so far using database primitives, opcode execution model, in memory database, kernel fusion, online transaction processing and data optimization. To mitigate the data transfer bottleneck, compression, differential updates and Nvidia’s Unified Virtual Addressing (UVA) feature are reviewed.

Section 2 provides background information on GPUs and challenges faced when trying to leverage the GPU. Section 4 review multiple methods to accelerate database queries. Section 5 compares their approaches.

Section 6 reviews and compares mitigation methods. Section 6.4 compares the mitigation approaches.

Section 7 summarizes the paper.

2. GPU BACKGROUND
CUDA is an API for applications written in the C programming language. A proprietary compiler transforms the CUDA program into Nvidia’s PTX assembly language. An application which is run using the GPU is called a kernel. The GPU contains many SIMD (single instruction multiple data) Streaming Processors (SP). Each Streaming Multi-processor (SM) contains 8 SPs. All SMs execute the same instruction, but operate on different data allowing for parallelism. Each thread is assigned to a kernel.

The primary obstacle to leveraging the GPU is the extra overhead incurred when transferring data from the host computer’s main memory to the GPU’s device memory. Secondary problems are the management of multiple virtual address spaces and synchronization between threads.

Figure 1 provides a simplified model of GPU and CPU memory interaction. Before a GPU can perform computations, the data has to be copied from the main memory on the host system to the global memory on the graphics card. The problem is twofold. First most implementations are limited from 1 GB to 4 GB per GPU [25]. For large data sets exceeding this limit, additional overhead is incurred when copying data between main memory and GPU memory.

Secondly the PCI Express (PCIe) bus between the graphics card and main memory used to transfer data is bandwidth limited. These bottlenecks apply to any application attempting to leverage the GPU.
2.1 Thread and Memory Hierarchy

Figure 2 displays the memory hierarchy within the GPU. The fastest and smallest quantity of memory are private registers available to individual threads. Threads can form blocks and multiple blocks form grids. Registers are private to each thread. Thread blocks use shared memory which is visible to all threads within the same thread block. Register and shared memory are considered local memory since they are closest to the SM. They are only tens of kilobytes (KB) in size. Shared memory cannot be shared between different thread blocks. Global memory is several gigabytes (GB) in size and is slower (higher latency) than local memory but can be shared between all threads and thread blocks. The term GPU global memory is used interchangeably with GPU memory.

During execution, thread blocks are sent to each the SM. The thread blocks are divided into what Nvidia calls a warp, which is a group of 32 parallel threads. The warps are then scheduled to be run in the SM. All threads in the warp start at the same address but can branch to execute independently.

3. GPU FEATURES AND UVA

CUDA supports memory mapping which is also known as “page-locked”, “pinned” or “zero copy” memory. It locks a specified amount of memory to prevent the operating system from paging out to disk. The “pinned” memory is then mapped into the GPU’s address space. Mapped memory reduces the overhead of transferring data back and forth between the CPU memory and GPU global memory. An evolution of this capability is Nvidia’s “Unified Virtual Addressing” (UVA) feature. UVA allows the address space from multiple GPU devices and CPU to be mapped into a single address space. This is illustrated in Figure 3.

The advantage of UVA over “zero copy” is it simplifies programming for the developer, especially in multi-GPU configurations because they do not have to manage multiple address spaces and copy data between the GPU and CPU address spaces.

The disadvantage of UVA is the CPU’s main memory has to be “pinned” by the operating system, i.e this prevents the memory from being paged out to disk and reduces the amount of main memory available to other applications on the system.

4. QUERY PROCESSING

A database query is typically processed using the steps below [8]:

1. Convert query into an internal form. For example a SQL query is converted into an internal representation called an operation code or “opcode”.
2. Convert to canonical form to represent multiple variations of the query in the most efficient form.
3. Choose candidate low-level procedures to implement. These are implemented via primitives. For example a SQL SELECT can be implemented using the filter primitive.
4. Generate query/access plans based on a cost criteria (e.g. size of tables, CPU usage, input/output, etc...) and choose the cheapest plan. Generating multiple access plans consumes resources, thus there is a threshold to limit the number of candidate plans generated. In some cases generating the optimal plan may take longer than executing a sub-optimal plan.

There are four different relational join algorithms:
Figure 3: UVA (bottom) compared to the old method (top) with multiple address spaces.

- Non-indexed nested loop join (NINLJ): The join algorithm uses two loops to join two tables. One loop iterates over table1, while another iterates over table2 for each row in table1. If a row in table1 matches a row in table2 then it is joined.
- Indexed nested loop join (INLJ): This algorithm is similar to NINLJ, except an index is used to speed up the lookup for a row.
- Sort merge join (SMJ): Sorts the inputs of table1 and table2, then for each row in table1 it scans through table2 for matching row(s).
- Hash joins (HJ): Hash values are created for each row in table1 and table2. If a hash value from table1 matches table2 then the row matches.

4.1 Relational Joins

He et al [12] create the set of GPU accelerated primitives below:

- Map: Applies a function to an input relation.
- Scatter: Performs indexed writes from a relation e.g. hashing
- Gather: Performs indexed reads from a relation e.g. probing hash table
- Prefix scan: Applies a binary operator to an input relation.
- Split: Partitions a relation based on a partition function e.g. hash partitioning
- Sort: Sorts a relation.

The primitives are used to implement a number of join algorithms. They compare their algorithm against the performance of the open source database MonetDB [21] and CPU optimized primitives for joins. GPU accelerated joins along with memory optimizations improve the performance of primitives 2-27x. When these primitives were used to implement relational joins, this resulted in a speedup of 7x for non-indexed nested loop join (NINLJ), 6x for indexed nested loop join (INLJ), 2.4x for sort merge join (SMJ) and 1.9x for hash joins (HJ).

4.2 SQLite

Bakkum and Skadron [4] provide a novel way of leveraging the GPU by using a SQL interface instead of the traditional method of implementing primitives such as map, scatter, gather, reduce, etc... to build queries. SQLite is a simple database designed to be embedded into the source code of applications such as the Firefox web browser or mobile applications. Its entire source code is inside a single file for easy integration into an application which requires a database. They modify the SQLite virtual machine to redirect op codes generated by the SQL compiler to the GPU.

Bakkum and Skadron [4] indicate mapping main memory to the GPU address space for Direct Memory Access (DMA) using Nvidia’s “zero-copy” feature is not feasible due to the low bandwidth without providing any details. Their citation from the older Nvidia CUDA Programming Guide v2.3.1 [23] is no longer available and the new version 4 guide [24] does not mention this limitation. It is possible they were comparing the relative bandwidth of the PCI Express (PCIe) bus to the internal memory bandwidth on board the GPU which is at least an order of magnitude greater and not feasible for their type of workload.

The paper indicates there is a pinned memory limit on Linux. The Linux manual pages for `ulimit -l` “The maximum size that may be locked into memory” [20] do not indicate a limit on the amount of locked or pinned memory. A search through the Linux documentation and the internet did not reveal any indications of a 4 GB operating system pinned memory limit.

Results demonstrate GPU accelerated SELECTs and aggregation functions on numeric data types are faster than the CPU and result in an average 35x speedup. Lock contention from returning result rows consumes a majority of transfer time as the number of result rows increases.

4.3 CUDADB

Lin et. al [19] and Chang et. al [6] implement an in-memory database (IMDB) called CUDADB to demonstrate its scalability against SQLite. SQLite is configured to run entirely as an in memory database. The CUDA Data Parallel Primitives library [7] was used to implement primitives to support the SQL-92 standard.

CUDADB was benchmarked against SQLite. Overall, CUDADB scales better than SQLite beginning with result set sizes from 0.784% to 1.926% of the total number of records. They demonstrate CUDADB’s execution time is independent of result set size.

4.4 GPUTx

He et. al [14] create an Online Transaction Processing (OLTP) execution engine accelerated by the GPU called
GPUTx for in memory databases. OLTP workloads consist of tens of thousands of small transactions consisting of random reads and updates. The authors create a bulk execution model to batch and execute the transactions together.

Each transaction was categorized into a transaction type and implemented as a stored procedure within the database. GPUTx achieves a throughput 4-10x higher than the CPU. The results exclude data transfer time and assumes the dataset fits entirely in GPU global memory.

4.5 Data Optimization

Govindaraju et al. [10] perform initial research into optimized data placement for GPUs several years before CUDA was released. Bakkum et al. [2] optimizes data placement by creating a data structure called a Tablet which is optimized for reads by the CPU and GPU. They also incorporate their earlier work [3,4] with SQLite’s opcode execution model and mapped memory.

Each Tablet is a fixed size which allows for coalesced memory access from the GPU, which is up to 10x faster. Their opcode execution model is also more efficient than using primitives because only one GPU kernel is needed as seen in Figure 4. By combining Tablet, opcode execution and memory mapping they were able to achieve approximately 4x speedup for integer and floating point data over multi-core CPUs.

Figure 4: Left: The opcode model requires only one GPU kernel. Right: Typical approach with primitives requires multiple GPU kernels [2].

4.6 Kernel Fusion

Wu et al. [29] propose Kernel Weaver, a data movement optimization for Relational Algebra (RA) operators used in the data warehousing context. Kernel Weaver fuses multiple GPU kernels to reduce the number of GPU kernels and intermediate data created. GPU kernel fusion is similar to loop fusion. Examples of RA operators include UNION, INTERSECTION, DIFFERENCE, CROSS-PRODUCT, JOIN, PROJECT and SELECT.

The two main benefits of kernel fusion (KF) are reduced memory footprint and larger compiler optimization scope. A smaller memory footprint decreases memory accesses to GPU global memory since intermediate data can be stored in GPU local memory. This reduces the bandwidth limited PCIe bus traffic allowing larger data sets to be processed. Temporal data locality is improved as well. The compiler has a larger optimization scope which allows for increased optimization versus optimizing the several kernels separately. For example, the compiler can remove redundant computations between multiple kernels.

To fuse RA operators, a dependence graph is constructed (Figure 5) using a greedy algorithm. Fusion criteria is determined by resource constraints in terms of shared memory and registers available to each Stream Multiprocessor (SM). Fusing operators in the beginning of the dependency gain the most performance benefit since these process the most data compared to subsequent RA operators.

Figure 5: Example of constructing dependence graph: (a) database program; (b) dependence graph [29].

KF achieved a 2.89x speedup in GPU computation and 2.35x speedup in PCIe transfer across the micro-benchmarks used.

5. QUERY ACCELERATION

There are three approaches to accelerating database queries using the GPU. The first and most commonly used method is to write a primitive to implement each SQL operator. For example GROUP BY and aggregate functions COUNT, SUM are implemented using the sort primitive. SELECT is implemented by the filter primitive [13]. Each primitive is implemented inside its own separate CUDA kernel. During query execution, the host system is used to manage multiple kernels. This approach is taken by several authors [13], [12] and [6].

The second approach is an opcode execution model. This approach is exemplified in Section 4.2 where SQLite’s SQL compiler opcodes were modified to send work to the GPU instead of CPU.

A third approach is kernel fusion. RA primitives are created to support queries as in the first method, however the multiple GPU kernels are fused together [29] to reduce resource usage and increase compiler optimization scope.

Implementation with primitives is easier due to the availability of the CUDA Data Parallel Primitives library [7] which was used by [6,19]. However the disadvantage is each primitive requires its own CUDA kernel, thus the CPU incurs additional overhead (e.g. synchronization) to manage multiple kernels [1].
The opcode execution model is very flexible because the opcodes can be modified for use with CPUs and GPUs. The disadvantage with Bakkum et. al [3, 4] is their implementation is limited to numerical data and SQL aggregation functions like COUNT, SUM, MAX, etc. There is no support for relational joins, GROUP BY and non-numeric data types like characters, timestamps and binary objects. They assume the entire data set fits inside the GPU’s global memory and any data transfer costs are amortized over many SELECT statements.

Bakkum et. al [2] integrate many of the features above by combining the opcode execution pattern with optimized data structures to efficiently transfer data and mapped memory to increase performance. Their approach also reduces multiple GPU kernels to one.

GPUTx batches multiple transactions and is applicable to OLTP workloads.

Previous studies indicate data transfer consumes 10-90% of total execution time [9, 13]. For all papers except [29], data transfer remains a serious bottleneck which is why some authors exclude the data transfer time [12] or ensure the entire data set is small enough to fit into global memory [4,6]. This was done to isolate and benchmark query performance from the effects of data transfer.

Unfortunately, no common software and hardware benchmarking platform was used which makes it next to impossible to make a direct performance comparison between the methods discussed above.

6. MITIGATION

6.1 Compression

The lowest level mitigation method is to apply data compression when moving data between the GPU global memory and CPU memory. Fang et. al [9] apply nine different compression algorithms to reduce the PCIe bandwidth required when moving data between main memory and GPU global memory. Previous studies have shown data transfer consumes 15-90% of total time when executing a query [9,13]. They trade extra computation consumed by compression and decompression for reduced bandwidth and memory usage.

Fang et. al implement a compression planner and compression aware query optimizer to pick the optimum compression scheme. They compare their results against native MonetDB database.

Using compression, throughput reached a maximum of 56 GB/s compared to approximately 6 GB/s for non compressed data on the PCIe bus. The performance of CPU based partial compression is similar to GPU based partial compression. Both methods are faster than native MonetDB for the TPC-H queries benchmarked. Partial compression performed the best because using full compression increases the probability the data has to be decompressed in order to evaluate the query. With table-scans, enabling compression is slower than no compression due to the large data set which needs to be transferred over the throughput limited PCIe bus.

One issue not addressed by Fang et. al [9] is the performance overhead incurred when inserting or updating data into the database since the data will have to be compressed before being stored in the database. For databases with heavy insert or update activity on large data sets, this may impact performance negatively.

Due to the close performance between partial CPU compression and partial GPU compression, in the future it may be worthwhile to investigate further optimization of compression algorithms for CPUs to determine how well they compare to GPU optimized algorithms. Paper [18] from Intel indicates the speedup for GPUs versus CPUs in benchmarks such as FFT, sort, etc. averages about 2.5x after optimization.

6.2 Differential Updates

Databases using dictionary compression perform well during reads, but suffer because the dictionary is recreated during updates/deletes. A technique called differential updates allows faster modifications by maintaining a delta storage. The delta storage is merged with the main storage to create an updated dictionary. Krueger et. al [17] propose a new method to accelerate the merge process using the GPU.

The size of the delta data and frequency of the merge must be carefully balanced. A larger delta increases the time consumed by reads and increase the length of the merge process. Frequent merges may also negatively impact performance.

Krueger et. al adopt the algorithm used in HYRISE [11] and use Nvidia’s Thrust [26] library for implementation. The HYRISE algorithm stores modifications to the database in its own dedicated buffer (delta storage) before asynchronously merging with the main database. The merge algorithm is described below:

1. Merge dictionaries.
   (a) Build dictionary by extracting unique values from the delta. This step is not a good candidate for GPU acceleration.
   (b) Merge main and delta dictionaries to create a sorted dictionary from unique values. Good candidate for GPU acceleration because each step can be run in parallel independently.

2. Update compressed values.
   (a) Compute new compressed value length required to represent the distinct values of the merged main and delta dictionaries from step 1 (b).
   (b) Update compressed values. Majority of runtime in this step, but due to sequential dependencies it is not good candidate for GPU acceleration.

Krueger et. al identify the steps in the differential update merge algorithm which can be run in parallel. The majority of the runtime is consumed in merge step 2 (b), but this cannot be optimized due to sequential dependencies. Since only merge step 1 (b) can be accelerated, the overall performance increase was limited. Step 1 (b) was accelerated 40%. Dictionary Sort Merge was 50% of CPU implementation throughput. 60% of the total execution time was consumed by data transfer.

6.3 UVA

UVA can be applied at a higher level by optimizing an algorithm’s design to reduce data transfers. Kaldewey et. al [15] investigate how UVA can be used to accelerate database partitioned hash joins. The authors port and optimize a multicore CPU hash algorithm [5] to the GPU.

A hash join is performed by creating a hash table for one of the smaller tables involved in the join known as the “build”
phase. The second phase called the “probe” takes place when trying to match each row in the larger table against the smaller table via the hash function. A partitioned hash join by Kim et. al [16] was also ported as well.

Using the pre-UVA method, the hash algorithm copies data between the main memory and GPU memory multiple times during the build and probe phases. For example copying a build table into GPU global memory, deleting it, creating a hash table in GPU global memory and so on. Since each set of memory is addressed via its own individual virtual address space there is also overhead in context switching when transferring control between the GPU and CPU.

With UVA enabled the hash table can be built in GPU memory while reading data from the CPU’s main memory. During the probe phase each row can be read from main memory and concurrently probe GPU memory. UVA eliminates the intermediate data transfers.

A limitation is the hash table must fit into the GPU global memory. Additional algorithms will have to be implemented to handle any hash table overflows if they do not fit into GPU global memory.

Kaldewey et. al simulate the worst case scenario in which the two tables being joined are identical in size with data uniformly distributed. They also control the rate of matches between the probe and hash table in order to evaluate the hash join. Conventional hash joins are 5x faster than the CPU implementation. Partitioned hash joins marginally benefit from UVA.

**6.4 Mitigation Summary**

Data compression [9] is a low level mechanism which can be applied to any application which needs to move data between the CPU’s main memory and GPU global memory. Differential updates optimization is limited to dictionary compression scenarios. Data structure optimization can also be applied to improve query performance.

UVA benefits applications designed to leverage multiple GPUs and need to map portions from the CPU’s main memory to GPU global memory.

We note that kernel fusion from Section 4.6 can also be considered a mitigation method since it can be applied to other domains (e.g. linear algebra), other representations (e.g. OpenCL, PTX, etc...) and other devices (e.g. CPU).

Unfortunately, no direct comparison can be made between the mitigation methods because there was no common benchmarking platform (software and hardware) used to measure their performance. Data transfer remains a bottleneck preventing full exploitation of the GPU.

**7. SUMMARY**

This survey reviews several approaches to accelerating queries. The first method is to create GPU accelerated primitives, the second is to use an opcode execution model. The third method also uses primitives, but fuses GPU kernels to reduce memory usage and increase compiler optimization scope. At a high level, OLTP workloads can be batched together to improve throughput.

The mitigation methods of compression, data structure optimization and UVA can be applied to non database applications. The differential update technique can be used to enhance databases using dictionary compression, but has limited impact.

**8. REFERENCES**


[27] OpenCL. Khronos Group OpenCL. http://www.khronos.org/opencl/.
