A Survey on Improved Performance of Database Related Operations using GPUs

Abstract

The growth of size in datasets and the demand for tools to perform scientific data-analysis have been rising in the recent years. Many database researchers are looking for ways to accelerate various database operations in order to maintain good or better performance with today’s larger datasets. A lot of the work to accelerate these operations has been done for parallel database systems that involve a loosely-coupled cluster. However, with the arrival of the GPU, and the techniques offered by the General Purpose Graphics Processing Units (GP-GPUs), new opportunities are offered for the database community. Some researchers are starting to use these devices to improve database operations using the relatively inexpensive tightly-coupled system of GPUs instead of the more expensive loosely-coupled systems found in pools of processors. These tightly coupled systems also offer additional advantages in that there is almost no overhead in creating threads and their main memory bandwidth is higher than a CPU’s. Many researchers in the database community have found ways to exploit this high thread-level parallelism and higher memory bandwidth in order to accelerate several database operations. This article presents an overview of the current techniques and algorithms used on GPUs in order to accelerate database operations.

Keywords: GP-GPU, high-performance in database operations, GPU, high thread-level parallelism.

# Introduction.

GPUs are both powerful and inexpensive devices that could be used in order to obtain more computational throughput. Not only can they perform faster than CPUs in number of float point operations per second but they also have a higher rate growth in their performance. Over the last decade the growth rate of GPU performance has been higher than that of the CPUs. Figure [x] shows how the NVIDIA’s GPU’s floating point performance has increased dramatically between the years 2003 and 2008, and how the gap in performance between the CPU and GPU has widened over those years. GPUs were primarily designed and used for performing algorithms around graphic computation, such as transforming, rendering and texturing geometric primitives such as triangles. However, many have sought techniques for using the GPU to perform tasks not related with graphics that are typically handled by the CPU.



Figure . Taken from [NVIDIA Programming guide]

Many researchers view GPUs as today’s most powerful computational hardware for the dollar. Researchers and developers in areas not related to graphics have also become interested in harnessing this power for general-purpose computing. This effort is collectively known as GPGPU (for “General-Purpose computing on the GPU”). [#:Survey Paer]

Two types of GPU programming languages have been used in the work discussed in this paper. There are graphics APIs such as DirectX, and OpenGL, and GPGPU languages such as CUDA. The graphic APIs process textures through a programmable hardware pipeline. To drive the computation vertices and pixel, processors are employed. These APIs allow one to directly utilize the hardware features related to rendering and visualization. In a lot of work that is presented, which was previous to the development of the GPGPU language CUDA, developers had to use the graphics APIs to map their programs to the graphics rendering mechanism, and so, such this work has been expressed in terms related to that programming. In order to clarify common terminology, an overview of important terms will be presented here. Often the terms kernels, and fragment programs can be used interchangeably, they both refer to a piece of code that we want to compute in parallel.

Even though GPUs are some of the fastest devices out there for computation, they are not suitable for many of the tasks that are commonly handled by the CPU. Currently, GPUs lack some useful computational constructs, such as support for various instructions. Also support for data types such as integers has recently arrived, but it has not been around for previous years of GPU research. They also do not commonly support 64-bit precision yet, and also sacrifice some performance in order to support it [3]. GPUs are also unable to replace CPUs in various types of applications where data dependency is so high that it is only possible to parallelize few tasks, or none of them. As a rule of thumb, if given an application one cannot create at least thousands of threads; there is no benefit in performance in using a GPU. Many applications are also dominated by both memory communication, and memory serialization effects associated with indirect memory addressing are also not a good match for GPUs. Finally, there are also some applications that computationally perform better in GPUs, but the cost of transferring the data from the CPU to the GPU would be greater than the benefit in performance obtained.

Most database operations are typically data intensive operations, and are not a good match for a GPU. However there are some of them that are also very computationally intensive and could benefit from the GPUs high-level parallelism. Among them are some operations that are performed in database queries, which could be accelerated when, streaming several queries of a dataset [#x]. Also some GPU-based sorting algorithms have better cost-effective performance than CPU-based algorithms [#x]. Join Operations are computationally expensive, and have been accelerated by using the GPU-based sorting algorithm in order to sort the records based on the join key [#x]. Finally query evaluation using indices may take advantage of the numerous threads in a GPU in order to examine several records of an index simultaneously in order to answer queries quickly [#x]. The rest of the paper will discuss each of these operation in detailed and is organized as follows. Section 2 provides a brief overview of the basic database operations that can be performed efficiently on the GPU.

GPU-based algorithms perform computations on 2D arrays of 32-bit floating point data values known as textures. Each array element corresponds to a pixel. Pixels are transformed by programmable fragment processors, each executing the same fragment program on each pixel. The multiple GPU fragment processors perform data parallel computations on different pixel arrays simultaneously. This simple data-parallel architecture avoids write-after-read hazards while performing parallel computations.

**Database operations when performing data queries**

Govindaraju et al. presented some novel algorithms for accelerating various database operations on the GPU. They considered algorithms that aimed at operations such as conjunctive selections, aggregations, and semi-linear queries. These operations are often essential in database, data warehousing, and data mining applications. They took advantage of the inherent pipelining and parallelism of GPUs. The single instruction and multiple data (SIMD) capabilities, and vector processing functionalities were also useful.

Their algorithms took into consideration some of the limitations of the programming model of current GPUs and perform no data re-arrangements. They compared their performance with an optimized implementation of CPU-based algorithms and their experiments indicated that the graphics processor available on commodity computer systems is an effective co-processor for performing database operations.

The database operations they focused on mostly revolved around select queries which were of the following form:

SELECT **A**

FROM **T**

WHERE **C**

Here, **T** is the table that is being queried. **A** could be a list of attributes or aggregations such as SUM, COUNT, AVG, MIN, MAX defined on individual attributes. **C** is the condition of the query which is defined as a Boolean combination of predicates. A boolean combination is a set of predicates combined using relational operators such as AND, OR, EXIST, NOT EXIST among others. Predicates have the form op where op is the operator and it could be any of the following: .

Essentially predicates, boolean combinations and aggregations are considered the three different categories of database operations which could be accelerated using GPUs.

To speed up predicates in the form of they were evaluated via a depth test and stencil test. To use the depth test for performing comparisons, attribute values need to be stored in the depth buffer. They use a simple fragment program for copying the attribute values from the texture memory to the depth buffer.

Predicates in the form of, are first transformed into a semi-linear query which is then evaluated as a predicate with a constant. This algorithm performs the operations using the vector processing units on the GPUs.

To solve Boolean combinations in the GPU efficiently they are first rewritten in a conjunctive normal form (CNF) omitting the NOT operators. Then the stencil test is then used repeatedly for evaluating a series of logical operators recursively with the intermediate results stored in the stencil buffer.

The Aggregations are all implemented using the counting capability of the occlusion queries on GPUs. Because the aggregations Min and Max are considered special cases of the kth largest number problem, a single algorithm was generated.

The algorithms for the database operations in the predicates, boolean, and aggregation categories were implemented on a GPU and their performance was compared with optimized implementations of CPU-based algorithms. Tests were performed by testing different types of queries that would use one or more of the database operations described in the three different categories. Predicate evaluation queries only involved queries were a predicate was included in the condition. Boolean combination database operations were tested through range since these can be expressed as a boolean combination of two predicates. For more complex boolean combinations of predicates, queries with more than one relational operator, multi-attribute queries were used. Finally, the algorithm kth largest number was tested for queries that involved the MIN and MAX aggregations, and an accumulator implementation in the GPU using a mipmap texture was used in order to test the SUM and AVG aggregations. The results of these tests are summarized in the following table [x].

|  |  |  |
| --- | --- | --- |
| **Type of Query or Database Operation** | **Approximate Speed up** | **Approximate Speed up (Computational Time only)** |
| Predicate Evaluation | 3 | 20 |
| Range Query | 5.5 | 40 |
| Multi-attribute Query | 2 | 20 |
| Semi-linear query | 9 | not mentioned |
| kth largest number | 2 | 3 |
| Accumulator | 0.05 | not mentioned |

Overall they conclude in their research that algorithms for semi-linear queries and selection queries had the highest performance gain when implemented in GPUs. A medium performance gain was observed for the kth largest algorithm and no performance gain was observed for the accumulator algorithm which ended up being 20 times slower than its CPU-based implementation.

# Sort operations.

The Sort operation in databases or datasets transforms an unordered set of data into an ordered set of data with a given criteria. Different techniques have been developed to sort on a CPU. However, most of these techniques are not good candidates for a GPU implementation because most of them have data-dependencies difficult to get around. For a sorting algorithm to be GPU-friendly, it has to be oblivious to the input data.

Algorithms for sorting on GPUs are based on sorting networks. A sorting network consists of a network of wires and comparator modules which are used to sort a sequence of numbers. Each comparator will connect two wires and compare two values. It will then output the smaller value to one wire and the larger value into another wire. Sorting networks sort input data in a fixed number of steps for any given data, meaning they are independent of the input data. They also have a fixed communication pattern, making gather operations possible (gather operations have been implemented in GPUs before). Given these two reasons sorting networks are good candidates for GPUs because they can be implemented without data-dependent branching unlike CPU algorithms. The computational complexity of a sorting network algorithm in a GPU is .

Many researchers have proposed variants of the sorting network algorithm to improve performance. Purcell et al. [35] describe the bitonic merge sort which uses a periodic balanced sorting network.

This algorithm takes unsorted data from an input array or texture, sorts it and put it back in texture memory. A fragment program is used for each step of the sorting network. The fragment program takes two fragment values from the input array or texture and then performs a compare-and-swap operation on the texture values, which are based on the sort parameters.

A bitonic sequence is a monotonic ascending or descending sequence. Given an input array, the bitonic sorting algorithm proceeds bottom-up, merging bitonic sequences of equal sizes at each stage. It first constructs bitonic sequences of size 2 by merging pairs of adjacent data elements where. Then bitonic sequences of size 4 are formed in stage 2 by merging pairs of bitonic sequences  and  where. The output of each stage is the input to the next stage. The size of the bitonic sequence pairs doubles at every stage. The final stage forms a sorted sequence by merging bitonic sequences  (see Figure [x]). [literally copied#x]

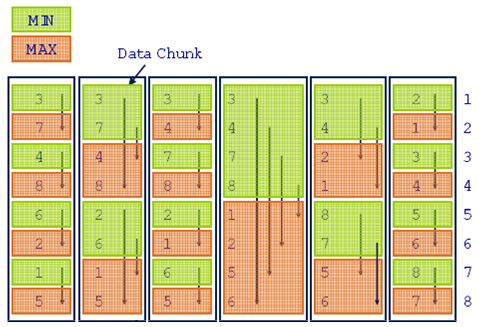


Figure . This figure illustrates the

bitonic sorting network on 8 different values

“Some sorting networks are implemented using the texture mapping and blending functionalities of the GPU. At each step of the sorting network, a comparator mapping is created at each pixel on the screen and the color of the pixel is compared against exactly one other pixel. The comparison operations are implemented using the blending functionality and the comparator mapping is implemented using the texture mapping hardware, thus entirely eliminating the need for fragment programs.”

Govindaraju et al. present the GPUTeraSort, a sorting algorithm developed to sort database rows based on keys. It is able to sort billion-record wide-key databases that do not fit in the GPU video memory or in the main memory using a GPU. Their algorithm uses the GPU as a coprocessor that uses its data and task parallelism to perform memory-intensive and compute-intensive tasks ­­­while the CPU is used to perform I/O and resource management. The GPUTeraSort has a task pipeline with two phases. In the first phase, it reads disk asynchronously and builds keys in the CPU. At the same time it sorts using the GPU and then it generates runs and writes back to disk. This first phase is illustrated in image [x]. The sorter phase in this algorithm makes use of the bitonic sort network, which will sort the data that is transferred from the CPU to the GPU. Finally, in the second phase it reads, merges and writes the runs, back to the CPU, completing the sort. [#x]

Govindaraju et al. also explain that part of the reason why current algorithms running on the commodity CPUs cannot achieve high sorting performance on such large partitions of data is because they incur in significant cache misses on datasets that do not fit the L1, L2, or L3 data caches, making it inefficient to sort partitions that are comparable to the size of main memory.

The GPUTerasort has five stages that were designed to be executed sequentially. However, some stages could be executed using multi-buffered pipeline-parallel independent threads:

The first stage involves the reader, which asynchronously reads the input file into a main memory buffer with size of 100 MB approximately. The reading bandwidth is improved by striping the input file across different disks so that the data is transferred from all disks in parallel. The I/O bandwidth and the CPU usage of the reader depend on the number of overlapping asynchronous I/O requests.

The second stage involves a Key-generator which computes the (key, record-pointer) pairs from the input buffer. In practice this stage is not computationally intensive but can be memory intensive it is because reading each key from main memory. It then sequentially writes a stream of keypointers pairs to the main memory of the GPU.

The third stage is the actual sorter which reads and sorts the key-pointer pairs. This stage is computationally intensive and memory-intensive on large buffers with wide keys.

The fourth stage is the reorder stage which rearranges the input buffer based on the sorted key-pointer pairs to generate a sorted output buffer (a run). On large databases, re-order is expensive because it randomly reads and writes long records from the input buffer and so it has many memory stalls.

In the fifth stage a writer asynchronously writes the run to the disk. Striping a run across many disks is not efficient fro Phase 2 reads; therefore the GPUTerasort cyclically writes the phase 1 runs to individual disks in very large transfers. The writer thread requires less than 10% of the CPU to achieve near-peak I/O performance.



Figure 3. First phase of the GPUTeraSort algorithm.

The performance of the GPUTeraSort was evaluated on billion-record files against optimized standard CPU-based algorithms. It was concluded that the overall performance of GPUTeraSort was in the mid-range GPU (costing around $300) which is comparable to that of a CPU-based algorithm running on a high-end dual Xeon processors (costing around $2,200). “In practice, GPUTeraSort achieves a good price-performance and outperforms the current PennySort benchmark.”



**Relational Joins on Graphics Processors**

We present a novel design and implementation of relational join algorithms for new-generation graphics processing units (GPUs). The most recent GPU features include support for writing to random memory locations, efficient inter-processor communication, and a programming model for general-purpose computing. Taking advantage of these new features, we design a set of data-parallel primitives such as split and sort, and use these primitives to implement indexed or non-indexed nested-loop, sort-merge and hash joins. Our algorithms utilize the high parallelism as well as the high memory bandwidth of the GPU, and use parallel computation and memory optimizations to effectively reduce memory stalls. We have implemented our algorithms on a PC with an NVIDIA G80 GPU and an Intel quad-core CPU. Our GPU-based join algorithms are able to achieve a performance improvement of 2-7X over their optimized CPU-based counterparts.

In summary, our GPU-based primitives and join algorithms achieve a speedup of 2-27X over their optimized CPU-based counterparts. We evaluated our join algorithms for both equijoins and non-equijoins, different data sizes, join selectivities and data processing algorithms, e.g., index searches, and need special care on the GPU. Existing techniques [40] for rewriting the branches on the CPU can also be applied to the GPU. This rewrite is especially useful for common and expensive operations. We acknowledge that this kind of rewriting in general is a difficult task for the run-time environment. Another example is that the synchronization mechanism for handling read/write conflicts, which happen constantly in query processing, is limited in the GPU. As a result, our primitives and join algorithms take extra computation such as computing the writing offsets to avoid the conflicts. This extra computation increases the work complexity of our algorithms by a constant factor. Second, with the exposure of the massively multi-threaded hardware architecture on the GPU, it also makes GPGPU programming trickier to ensure correctness and to fully utilize the essential GPU features such as data parallelism than the previous GPUs. In our work, we have developed a small set of primitives that are carefully designed and highly tuned for GPU join processing. Similarly, GPGPU programmers could produce better and faster programs using a set of well-defined primitives as building blocks to address this issue. Third, even though the latest GPU frameworks, such as CTM and CUDA, are a significant leap from the traditional GPUs in providing great details about the hardware architecture, they are still far behind the CPU vendors' tradition of giving sufficient details about the hardware specification, e.g., the memory hierarchy. Currently, we mainly rely on empirical experiments to estimate the hardware parameters and to identify the suitable settings for our algorithms. Fourth, the power consumption of the GPU is higher than that of the CPU. In our experiments, the GPU requires a power supply of 450 Watts, whereas the CPU requires 95 Watts only. It is desirable to develop software or hardware techniques to reduce the power consumption of the GPU.

Finally, as a co-processor, the GPU requires advanced software techniques to support complex workloads. For example, lacking hardware support for complex data types is an inherent weakness of the GPU. Currently, we can use software solutions for supporting more complex data types such as high precision numbers on the GPU [38]. Fortunately, GPU vendors plan to support high precision numbers such as double in the near future.

Graphics processors have become an attractive alternative for general-purpose high performance computing on commodity hardware. The continuing advances in hardware and the recent improvements on programmability make GPUs even ore suitable for database query processing than before. In this study, we have designed a small set of data-parallel primitives for relational join processing on GPUs. These primitives provide high-level abstractions for data-centric operations and are highly tuned to fully utilize the architectural features of graphics processors. We have implemented four representative relational join algorithms using these primitives and have compared the join performance with optimized CPU-based in-memory join algorithms. We find that our GPU joins achieve a speedup of 2-7X over their optimized CPU-based counterparts. This paper focuses on GPU join processing in the video memory. We believe this is an important but initial step towards building a high-performance, general-purpose database query processor using the GPU. One interesting future direction is to evaluate our join algorithms with more complex workloads. Additionally, we are interested in how to schedule the execution of relational query processing between the GPU and the CPU so that their computation power is fully exploited.

Query Evaluation when using Indexing.

Luke Gosink et al. proposed a new data structure that reduces the size of the data that needs to be transferred from the CPU to the GPU. They use a binning mechanism in order to encode the data such that it takes less space. In their paper they refer to the actual data from the projected column as high resolution data, and the encoded data as low resolution data. To illustrate this idea with an example, take arbitrarily a column of integers for example where each element is four bytes long in a 32-bit Intel architecture. When each integer is encoded using their binning scheme, each element is reduced to occupy only one byte, that in all saves three bytes of space for each element that needs to be transferred to the GPU. In this case, our original integer values would be the high-resolution data, and our encoded values would be the low-resolution data. Because by changing the encoding one cannot completely answer a given query, a second data structure is required to resolve the query where the encoded data is not enough to determine which attribute values fulfill and do not fulfill the condition. For these boundary bin values, they use a data structure that contains the high resolution data called the Data Parallel OrBic Structure. With this structure they carry out a candidate-check and effectively resolve the query.



What is obtained in the end, is an encoded data table, which is also called low resolution data.

and They name this structure the Data Parallel OrBiC structure. This structure is composed of encoded data tables and an OrbiC. Ref: Luke J. Gosink

**Conclusions**

A survey of the current algorithms that have been implemented in GPUs is presented.

One of the conclusions of this paper, is that one of the greatest bottlenecks in applying GPUs to database operations, in particular the case where query processing is involved, is the fact that most of the time is spent transferring data from the CPU to the GPU.

The GPU is viewed by many of the database researchers as an effective co-processor to the CPUs. While it is clear that the GPU can outperform the CPU in some database operations they are not a complete substitute for the CPU, and only some operations may be accelerated with a GPU.

It is also important to add that a lot of the research may not affect databases directly but they provide primitives that can be used for databases, such as sorting primitives or search primitives.

Also GPUs are advancing. There is more that can be done with every fragment or multiprocessor. Many of them are starting to offer operations they didn’t offer before because they are advancing so a lot more algorithms will be possible to implement in GPUs as well.

References

1. Luke J. Gosink, Kesheng Wu, E. Wes Bethel, John D. Owens, Kenneth I. Joy: Data Parallel Bin-Based Indexing for Answering Queries on Multi-core Architectures. SSDBM 2009: 110-129

2. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M.C., Manocha, D.: Fast computation of

database operations using graphics processors. In: Proc. of SIGMOD. (2004) 215–226

3. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graph-

ics co-processor sorting for large database management. In: Proc. of SIGMOD. (2006) 325–

336

4. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26 (2007) 80–113

Context switch is not as expensive as in CPUs