Boosting an Embedded Relational Database Management System with Graphics Processing Units

Samuel Cremer, Michel Bagein, Saïd Mahmoudi and Pierre Manneback
Computer Science Department, University of Mons, Rue de Houdain 9, 7000, Mons, Belgium

Keywords: In-memory Database Systems, Embedded Databases, Relational Database Management Systems, GPU.

Abstract: Concurrently, with the rise of Big Data systems, relational database management systems (RDBMS) are still widely exploited in servers, client devices, and even embedded inside end-user applications. In this paper, it is suggest to improve the performance of SQLite, the most deployed embedded RDBMS. The proposed solution, named CuDB, is an "In-Memory" Database System (IMDB) which attempts to exploit specificities of modern CPU / GPU architectures. In this study massively parallel processing was combined with strategic data placement, closer to computing units. According to content and selectivity of queries, the measurements reveal an acceleration range between 5 to 120 times - with peak up to 411 - with one GPU GTX770 compared to SQLite standard implementation on a Core i7 CPU.

1 INTRODUCTION

In recent years, to deal with the exponential growth of data volumes, numerous new database techniques have appeared. Major improvements have been accomplished, especially in the area of Big Data systems. The different issues involved in current data growth do not only concern datacentres but also end-user applications. Whether with desktop platforms or mobile devices, numerous end-user applications embed an RDBMS - such as SQLite, SQL Server Compact or MySQL embedded. Such embedded RDBMS usually serve as storage systems, as well as cache systems to reduce the number of interactions between clients and servers, and hence preserve low latencies of user interfaces. With mobile systems, they are also used to compensate for long periods of connection loss. Increasing client-side computing capacities enables the processing of larger data volumes and hence to reduce the number of client-server communications.

In this paper a hybrid implementation over CPU and GPU is suggested in order to improve SQLite performances, which is the most widely deployed database engine throughout the world[1] (it is part of the majority of smartphone OS, browsers, Dropbox clients, etc.). Standard implementation of the SQLite engine is purely sequential. Its performances can be improved by using all processing units of CPUs and GPUs. For numerous applications, GPU architectures are currently more efficient than CPUs (Huang et al., 2009) and have become essential in modern systems, even in small devices like smartphones. The major contribution of this paper is to propose a data placement strategy allowing the exploitation of a clever parallelism offered by multicore CPU and GPU architectures. The benefit provided by the proposition here is the improvement of the responsiveness of client applications and the energy efficiency of full "client-network-server" chains.

The remainder of this paper is structured as follows: Section 2 presents the state of the art and the CuDBs position. Section 3 describes the internal architecture of the system, its storage engine and how join queries are processed. Evaluation results are presented in Section 4, and this paper ends with the conclusion and outlooks.

2 STATE OF THE ART

The idea of using hybrid CPU/GPU architectures to accelerate the data processing of database engines emerged in 2004 (Govindaraju et al., 2004), some years before the release of general-purpose processing on GPU (GPGPU) frameworks. Two main approaches have been proposed.

[1] SQLite : Most Widely Deployed and Used Database Engine, *www.sqlite.org/mostdeployed.html*

Cremer, S., Bagein, M., Mahmoudi, S. and Manneback, P.
Boosting an Embedded Relational Database Management System with Graphics Processing Units.
DOI: 10.5220/0005995701700175
In *Proceedings of the 5th International Conference on Data Management Technologies and Applications (DATA 2016)*, pages 170-175
ISBN: 978-989-758-193-9

The first one emerged in 2007 with GPUQP (Fang et al., 2007). This approach divides query plans into different action patterns which could be processed with different levels of parallelism, either on CPU or GPU targets. GPUQP introduced the basic design architecture for most works which followed. So far, the vast majority of research in this field is focused on very specific aspects of DBMS, and does not provide a complete database engine. For example, GPUTx, proposed by the authors in (He and Xu Yu, 2011), focused on transaction management and their locking mechanisms. OmniDB (Zhang et al., 2013), is another system in which the authors paid more attention to the maintainability properties of source code. The objective of GPUDB (Yuan et al., 2013), was to analyse the abilities of GPUs for online analytical processing (OLAP). Ocelot (Heimel et al., 2013) can also be mentioned, an extension of MonetDB, a kernel-adapter approach to make a portable database engine across different hardware architectures, and CoGaDB (Breßet al., 2013) which is mainly designed to study the generation of execution plans. From what is known, the latest project which is closest to a DBMS is Galactica (Yong et al., 2014), but with a partial support of SQL, it is rather intended to be used in Big Data environments.

The second approach, initiated by Sphyraena (Bakkum and Skadron, 2010), forces full query plan processing on the GPU side. Sphyraena mainly suffers from numerous data exchange penalties through PCI Express bus and does not exploit CPU's parallelism, but it seems to be more promising in terms of speed and efficiency improvements for embedded databases.

Most previous solutions are partial DBMS and work with "read-only" databases. These different works are more targeted to Big Data systems and do not encounter many of the issues of full relational database managements systems.

With CuDB, it is suggested to boost embedded RDBMS running inside end-user applications, which is fully justified by size limitations and the lack of extensibility of GPU memories. The aim is to improve the performances at the RDBMS engine level, which implicitly increases the responsiveness of applications, while leveraging capabilities of available hardware architectures. CuDB is a hybrid CPU / GPU fully read-write RDBMS engine. The proposal targets a high performance solution for either server clusters, personal computers (workstations, laptops) or small devices (embedded systems). It can also be noted that only two commercial products, GPUdb (GIS-Federal, 2014) and Parstream (Hummel, 2010), use database engines on hybrid architectures (numerous CPUs and GPUs) but these are mainly oriented for geographic information systems (GIS) and Big Data environments.

3 DESIGN OF CUDB

3.1 Internal Architecture

Before understanding how the solution works, the architecture of SQLite will be presented briefly. SQLite can be subdivided into four logical units. The first unit is the interface where the incoming SQL queries are received and results are sent back to user application. The second logical unit is the SQL Command Processor where incoming SQL queries are parsed and compiled to a resulting query plan (opcode list). These opcode lists are comparable to an assembly style instruction list and are interpreted by the third unit of SQLite named "Virtual Database Engine" (VDE). This virtual machine is in charge of executing those opcodes on data stored and managed into the last unit of SQLite, the Storage Engine (SE). In CuDB, the SQLite API is preserved. The two first stages are also preserved in order to maintain SQL language support and to remain compatible with existing applications.

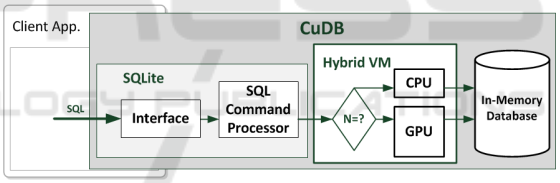

Figure 1: Internal architecture of CuDB.

VDE and SE units are the two components that intensely require the most resources. To exploit specificities of modern hardware architectures, CuDB embeds its own redesigned VDE and SE. (Figure 1). The VDE of CuDB is designed as a Hybrid Virtual Machine (HVM) which incorporates two distinctive processing engines. One is dedicated to the GPU, and the other to the CPU, but both are based on the same parallel paradigm: each thread processes the query plan on its own data rows. The workflow through the two first stages does not differ from the SQLite implementation. Then, the system adapts the received query plan in order to launch it for processing, either on the CPU or GPU. To get some benefit from the hybrid parallel hardware architecture, a same query-plan is processed by each CPU or GPU thread on a different dataset. Once a result row is calculated by a GPU thread, it is sent to the main memory through an asynchronous memory (pinned-memory). This technique allows threads to start processing next rows without

waiting for the end of the transfers, which increases the overall system performances.

To get some benefit from the high memory bandwidth between the graphic memory and GPU, the entire database is hosted directly by the GPU global memory (In-Memory-DB). The main advantage of this design is to avoid most of the data transfers between the CPU and GPU, and delete most of the unnecessary transfer latencies. With the "in-memory" database and with the majority of the extraction queries, experiments have revealed that performances depend primarily on the available memory bandwidth rather than computation power. Bandwidth between the GPU and its dedicated memory is often higher than that of a central RAM and CPU, and even more than PCI-Express links. This fully justifies the usage of GPUs for query processing.

As with the majority of GPGPU systems, a certain amount of data is required for processing before the GPU becomes more efficient than a CPU. To make an efficient use of hybrid architectures, HVM chooses to execute processing, either on CPU cores or GPU cores according to the data volume they have to process. The implementation of the multi-core CPU engine is based on the same principle as the GPU version, but using POSIX threads instead of the CUDA framework. To ensure maximum performance on CPU, table duplications are also kept in the main memory. The entire database is not duplicated in the central RAM memory, but only in tables which are processed faster on CPU (tables with less than 1000 rows). With the first implementation of CuDB, a separate memory management was used, with RAM and GPU memory. The "unified memory" method provided by CUDA (from version 6) was tested to avoid keeping a permanent copy of some tables. With the "unified memory", CPUs and GPUs use a same pointer to access the data. This makes writing code much easier since automatically managing, different memory transfers is achieved by the driver. However, it was noted, over various experiments, that severe slow-downs (between 2x to 9x) are introduced by the "overhead" of automatic memory management: the idea of using "unified memory" on the engine was abandoned. The work presented in (Landaverde et al., 2014) also concludes that using "unified memory" can usually cause performance degradations.

3.2 Storage Engine

SQLite is one of the rare RDBMS which features a dynamic typing system for each value: this is called the "Affinity" mechanism. To preserve compatibility with existing applications, CuDB supports dynamic data typing. However, such functionality entails a fairly high increase in complexity of treatments (dynamic casting of all data), and makes memory accesses less consistent (customized size of all data). Performances of GPGPU solutions are very sensitive to coherency of memory accesses (van den Braak et al., 2010), which makes CuDB efficiency and performance suboptimal when it handles dynamically typed columns. To reach the best performance, a selector was implemented for three different storage engine configurations:

- (1) Affinity: default storage configuration. It supports only dynamic typing similar to SQLite. Each tuple is always preceded by a header, which is required to support the "Affinity" mechanism. Records are stored in a linear way inside the memory, allowing high data compactness.

- (2) Strict: storage configuration with only static typing support. No longer need to store typing headers for each record. The performance is better through more consistent memory access, and by disabling "check type"' features. Like the "Affinity" mode, this is a row-oriented data structure.

- (3) Boost: to reach GPU peak performance, values are stored contiguously in the memory in order to provide coalesced memory accesses. This can be achieved thanks column-oriented and statically typed storage configuration.

Like MySQL, each table of the database can use distinguishing storage configurations. With these different storage engines, the database can be adjusted to its context while boosting the performances of applications with static data typing. With CuDB, database insertions do not block the entire table and are processed asynchronously by the CPU. On each database update, data persistency is provided by a "write-only" mirror database saved on the hard drive.

3.3 JOIN Queries

The processing time of join queries highly depends on the selected storage engine, for this reason, a description follows in this subsection of how CuDB handles JOIN clauses. As CuDB conserves the query compiler from SQLite, the query plans generated by the SQL Command Processor have to be used. The challenge is to find the best way to automatically parallelize the proposed plan. The query plan generated by SQLite for a join-query with non-indexed columns, like in query (1), proposes the creation of a temporary indexation structure (B+ tree) for inserting records of table $t2$. For each record of $t1$, corresponding records of $t2$ are searched inside the transient indexed structure.

(1) SELECT * FROM $t1$ JOIN $t2$ ON $t1.col2$=$t2.col3$

This query plan has a complexity of $O(m.log(m))$ for creation and filling, plus $O(n.log(m))$ for parsing the data. The classical B+ tree data structure used by the SQLite virtual machine is not optimal for an efficiently implementation on GPU architecture. To benefit from the massively parallel architecture of a GPU, a data structure that can be filled concurrently by numerous threads is required. Another constraint is that each thread makes its own search so that the traverse complexity of the structure must be adapted to a sequential search. Several existing GPU B+tree structures were investigated, for example, T-trees and CSS-tree structures. However, it was found that these structures cannot efficiently be filled in a parallel way. Moreover, with the GPU B+tree, a sequential pass through is slower than with a simple binary tree. Given that, once the temporary index is filled, the structure is used in read only mode, the approach is to use a simple vector of records. This record-vector can be filled in a parallel way by all the GPU threads, and can then be sorted, also in a parallel way. A sorted vector was obtained where each thread could make a dichotomic search in $O(log(n))$ operations.

Several GPU sorting algorithms were investigated and one of the fastest algorithms was the radix sort, with a time complexity of $O(n.w)$, in most cases, where n is the number of keys of word length w. If all keys are distinct, w is at least equal to $log(n)$, but the size of w can greatly increase when keys are strings. With a database usage, the performances of a radix-sort can be very variable depending on the key complexity. For a database engine, the preference was to select an algorithm which is independent of the sorted values and datatypes. This is the reason why a bitonic sorting algorithm was implemented. Figure 2 shows the behaviour of such sorting algorithm.

The bitonic sorter is a sorting network with a worst case complexity of $O(n.log(n)^2)$. The complexity of the bitonic sorter is slightly worse than with a radix sort, but the performances are stable regardless of the complexity of the sorted keys. To reduce the number of synchronizations, some implementation optimization techniques were carried out following (Hagen et al., 2010).

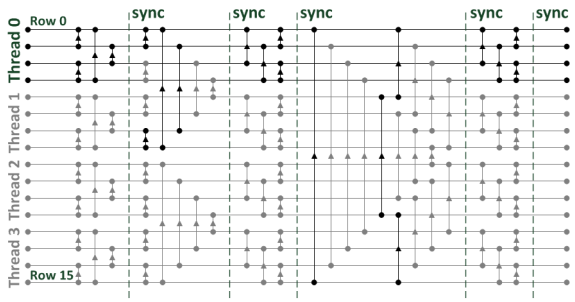

Figure 2: Implementation of the bitonic sorter algorithm on GPU. Each thread sorts 4 rows, and thread synchronizations are required after every 4 comparisons. The bold comparisons are processed by thread 0.

4 EVALUATION

For the performance evaluations, the time required to process two sets of SELECT queries (single table queries and JOIN queries) was measured with tables of varying sizes (between one hundred and one million rows with several numerical values and 80 character strings). The execution time of prepared statements was measured so that the compilation time of queries was not taken into account. The transfer times required to send the query plans to the GPU were considered, as well as the times needed by the GPU to send the results to the CPU. Different configurations of CuDB were compared with SQLite 3.8.10.2 and MySQL Embedded 5.7.11, both using in-memory databases. The experiments were run on a desktop with a 4-core Intel i7-2600K CPU with Hyper Threading, a 384-core GT740G5 GPU, and a 1536-core GTX770 GPU. The maximal available memory bandwidths were: 21 GB/s for the main DDR3 memory, 80 GB/s for the GT740, and 224 GB/s with the GTX770. As performances can slightly fluctuate, each test was done a hundred times. The behaviour of the system was quite constant, and for better readability of this document, the average values are presented here.

4.1 SELECT WHERE Queries

The different queries of this evaluation were applied to non-indexed tables with columns of various data types (numerical and strings). Figure 3 shows the average speedups obtained by the different test configurations with a standard implementation of SQLite as reference engine. The performances of the three storage engines were measured , but in order to not overload this paper, only the slowest "Affinity" and the fastest "Boost" storage mode are shown. With the hybrid engine of CuDB, queries applied to tiny tables were processed by CPU cores and when the table size exceeded one thousand records, the CuDB switched from the CPU to GPU processing engine.

For tables of one million records, and with the GTX770 GPU, speedups of 117x in Boost mode, 101x in Strict mode and 84x with the "Affinity" storage engine were obtained. As the boosting of the

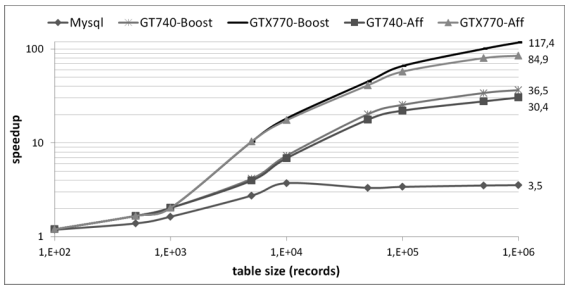

Figure 3: Average speedups with SELECT queries.

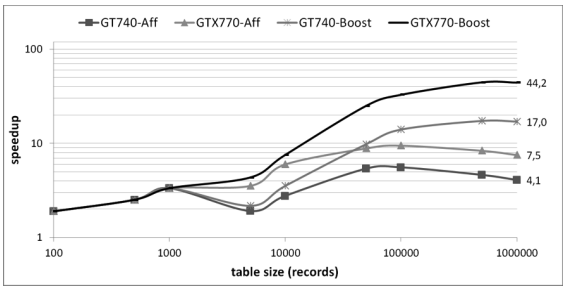

Figure 4: Average speedups with JOIN queries.

client-side embedded RDBMS system was investigated, the system was evaluated with an entry level GPU. A ”modest” GT740 procured already substantial speedups of 36x, 31x and 30x with the three different storage setups. These impressive speedups needed to be put into perspective; since SQLite is not the fastest in-memory RDBMS when scanning single tables. The ”memory” storage engine of MySQL 5.7 is more than 3 times faster than SQLite for large table scans. CuDB is still 33 times faster when comparing it to MySQL. Note that, during the experiments, the highest speedups were obtained (411x with the GTX770 and 107x with the GT740) for queries applied on fixed size string columns, and with a ”WHERE col LIKE susbstring%” search criterion.

4.2 SELECT JOIN Queries

In the previous subsection, it was shown that GPUs are extremely fast at processing full table scans thanks to their high memory bandwidth. In this subsection the results for JOIN queries will be presented. The subset of queries used for this evaluation includes multi-table join queries (with up to five involved tables) and also ”self-join” queries. The tables were not indexed and the join conditions were done on numerical data. The join queries were always done in tables with the same number of records. The average results are shown in Figure 4. Like the previous evaluations, for better readability, the results of the ”Strict” storage engine are not shown.

For tables of one million records each, the GTX770 GPU achieved an average speedup of 44x in Boost mode, 21x in Strict mode and 7,5x in ”Affinity” mode while the GT740 GPU obtained speedups of 17x, 8x and 4x. Unlike single table scans, the performance gaps between the different storage engines were significant. For large datasets, switching from ”Affinity” to ”Strict” makes join queries more than 2 times faster, and switching from ”Strict” to ”Boost” produces again a 2x gain. As explained in subsection 3.3, for each table join, the GPU has to perform a parallel sort. With ”Strict” tables, GPU threads do not have to check each type of each data and with ”Boost” tables, the memory accesses are coalesced. During sorting operations, each GPU thread has to access and compare multiple tuples several times, unlike single table scans where each thread accesses only one record. That is why the performances of join queries were impacted by the choice of an appropriate storage engine. Moreover with the need to access multiple records inside a single thread, the sorting algorithm needs global synchronizations. With current GPU architectures, there is no robust way to implement global synchronization without using the CPU. With CuDB, most synchronizations imply a save and restore of the GPU execution context. This is also the reason why general speedups obtained with join queries are lower than with single table scans.

The CPU-GPU engine switch still occurs at one thousand records, but with join queries, the GT740 becomes faster than the CPU when involved tables count a minimum of ten thousand records. That is why there is a slight performance drop at five thousand rows on Figure 4. Note that the peak speedups were obtained with ”self-join” queries (66x with the GTX770 and 28x with the GT740).

The results with MySQL are not shown on Figure 4 because MySQL was always much slower than SQLite with the set of join queries. As was explained in section 3.3, to reduce the time complexity for processing big datasets, SQLite uses transient indexes to compute join queries. MySQL does not, and implements such operations as nested loops. This results in multiple days of computing time for joining tables of a million records, while CuDB needs less than a second.

4.3 Energy Efficiency

As with the performance tests, two query categories were considered: single table queries, and join queries. Based on energy consumption measurements of all the queries with every platform configuration, for the following short report, an average of all energy efficiency ratios is proposed. Energy efficiency

is defined as a ratio of energy consumed by SQLite over energy consumed by tested platform. Figure 5 shows the average calculated energy efficiencies.

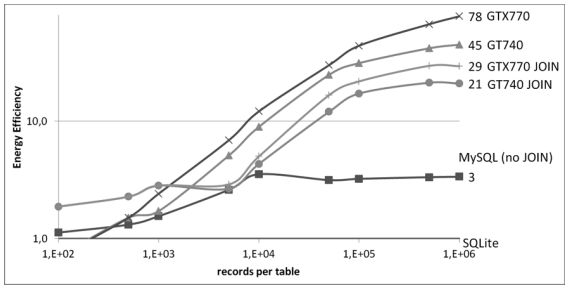

Figure 5: Energy efficiency: higher is better.

For a better readability, only results obtained with CuDB in "Boost" configuration are shown. These results confirm that the energy efficiency of embedded RDBMS can be significantly boosted by using a hybrid CPU/GPU query processing engine.

5 CONCLUSION AND FUTURE WORKS

In this paper, it has been shown that GPU architectures can be exploited to speed up processing of RDBMS. CuDB, an embedded RDBMS that is a performance upgrade of SQLite in the context of parallel hybrid architecture, has been presented. CuDB preserves the SQL support of SQLite as it retains its API. It has also been shown that speedups of more than 411x for queries on tables containing a million entries were achieved. Here the different measures have also shown that it is not necessary to have the most powerful GPU to achieve satisfactory accelerations. Weaknesses of GPGPU solutions for processing small amounts of data were also tackled by using a hybrid engine where light treatments remained on the CPU. As perspectivess, the support of some additional SQL clauses will be considered in order to be compliant with TPC-H and SSB benchmarks. A port of CuDB on OpenCL is also planned to target other GPU manufacturers. Another important challenge is to overcome the limitations of the GPU memory capacity which is currently limited to 16GB for high end GPUs. To overcome these size limitation, the proposal is to pipeline the query processing engine in order to mask memory transfers, and to transfer the data through circular buffer mechanisms. The overhead of transient memory requirements involved in complex join queries could also be larger than the physical GPU memory size. This will be also tackled by pipelining and circular buffering.

REFERENCES

Bakkum, P. and Skadron, K. (2010). Accelerating sql database operations on a gpu with cuda. In *3rd Workshop on GPGPU*, pages 94–103, Pittsburgh, USA.

Breß, S., Siegmund, N., Bellatreche, L., and Saake, G. (2013). An operator-stream-based scheduling engine for effective gpu coprocessing. *ADBIS*, 8133:288–301.

Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N., Luo, Q., and Sander, P. (2007). Gpuqp : query co-processing using graphics processors. In *SIGMOD/PODS'07*, pages 1061–1063, Beijing, China.

GIS-Federal (2014). Gpudb - a distributed database for many-core devices. 54th HPC User Forum, Seattle.

Govindaraju, N., Lloyd, B., Wang, W., Lin, M., and Manochad, D. (2004). Fast computation of database operations using graphics processors. In *SIGMOD/PODS'04 international conference on Management of data*, pages 215–216, Paris, France.

Hagen, P., Schulz-Hildebrandt, O., and Luttenberger, N. (2010). Fast in-place sorting with cuda based on bitonic sort. *Parallel Processing and Applied Mathematics*, 6067:403–410.

He, B. and Xu Yu, J. (2011). High-throughput transaction executions on graphics processors. *VLDB Endowment*, 8(5):314–325.

Heimel, M., Saecker, M., Pirk, H., Manegold, S., and Markl, V. (2013). Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720.

Huang, S., Xiao, S., and Feng, W. (2009). On the energy efficiency of graphics processing units for scientific computing. In *IPDPS'09*.

Hummel, M. (2010). Parstream - a parallel database on gpus. GTC2010, San Jose, CA.

Landaverde, R., Zhang, T., Coskun, A., and Herbordt, M. (2014). An investigation of unified memory access performance in cuda. In *HPEC 2014*, Waltham, MA.

van den Braak, G., Mersman, B., and Corporaal, H. (2010). Compiletime gpu memory access optimizations. In *ICSAMOS 2010*, Samos, Greece.

Yong, K., Karuppiah, E., and Chong-Wee See, S. (2014). Galactica: A gpu parallelized database accelerator. In *2014 International Conference on Big Data Science and Computing*, Beijing, China.

Yuan, Y., Lee, R., and Zhang, X. (2013). The yin and yang of processing data warehousing queries on gpu devices. *VLDB Endowment*, 6(10):817–828.

Zhang, S., He, J., He, B., and Lu, M. (2013). Omnidb: towards portable and efficient query processing on parallel cpu/gpu architectures. *VLDB Endowment*, 6(12):1374–1377.