Computer Science • 14 (2) 2013 http://dx.doi.org/10.7494/csci.2013.14.2.243

Marcin Pietroń
Paweł Russek
Kazimierz Wiatr

ACCELERATING SELECT WHERE AND SELECT JOIN QUERIES ON A GPU

Abstract *This paper presents implementations of a few selected SQL operations using the CUDA programming framework on the GPU platform. Nowadays, the GPU's parallel architectures give a high speed-up on certain problems. Therefore, the number of non-graphical problems that can be run and sped-up on the GPU still increases. Especially, there has been a lot of research in data mining on GPUs. In many cases it proves the advantage of offloading processing from the CPU to the GPU. At the beginning of our project we chose the set of SELECT WHERE and SELECT JOIN instructions as the most common operations used in databases. We parallelized these SQL operations using three main mechanisms in CUDA: thread group hierarchy, shared memories, and barrier synchronization. Our results show that the implemented highly parallel SELECT WHERE and SELECT JOIN operations on the GPU platform can be significantly faster than the sequential one in a database system run on the CPU.*

Keywords SQL, CUDA, relational databases, GPU

1. Introduction

Because of the growing interest in developing a lot of non-graphical algorithms on GPU platforms a NVIDIA CUDA framework was created [15]. The CUDA programming model makes GPU development faster and easier than low-level languages such as PTX, but still understanding the different memory spaces, interthread communication or how threads and threadblocks are mapped to the GPU are crucial to achieve optimized software. All of our implementations are done in the CUDA framework.

The SQL language is a well-known declarative language to operate on databases. It is a kind of connection between procedural programs and data stored in databases. Implementing SQL operations in the GPU enables database developers to accelerate operations with little changes in the existing source code.

There are limitations of GPU platforms that affect the possible boundaries of speeding up database operations. The two most important are memory size and host to GPU data transfer. The Nvidia Tesla m2090 [12] card used in our research enabled transfer which was about 4 GB/s. It has 512 cores inside, 5 GB device memory and peak efficiency about 655 GFlops. The second disadvantage is the size of memory. The current GPU's have a maximum of 4 gigabytes of global memory. In this size of memory only a fraction of the database can be stored. Therefore, it is an important task to monitor the size of data sent to the GPU on which SQL is executed. It should be added that in this size of memory not only the input data must be stored but also the output of the operation must be fitted in the GPU's memory space. Despite these constraints research shows that highly parallel SQL operation execution on the GPU very often outperforms the sequential CPU query execution.

The main goal of our work is to implement an interface for the SQL operations which will be executed on the GPU platform. Additionally, we want to build an SQL profiler which will be able to estimate the time of execution of the query and to check if there is enough free memory space to store the data. Our project is based on the ANTLR tool for parsing SQL queries. Then, queries are transformed to an internal form and sent as parameters to the GPU. The framework is independent but can be added to standard SQL databases by external functions. It should be added that the translation of the data from commercial databases to specific internal data creates a delay in the execution. This problem will be described further in this paper.

There has been a lot of research done in the data mining domain. The proof of the GPUs advantages and computational power. There are a wide spectrum of data mining algorithms implemented on a GPU such as binary searches, p-ary searches [5], k-means algorithm [7]. The two most closely researched are *Relational Query Coprocessing on Graphics Processors* [4] and *Accelerating SQL operations on a GPU with CUDA* [11]. The first one is about the implementation of the database but without direct SQL access. The second one is similar to our implementation. It consists of several SELECT WHERE clauses and aggregation operations.

2. The architecture of a GPU

The architecture of a GPU card is described in Figure 1. The graphical processor unit has a multiprocessor structure. In Figure 1 is shown N multiprocessor GPU with the M cores each. The cores share an Instruction Unit with other cores in a multiprocessor. Multiprocessors have special memories which are much faster than global memory which is common for all multiprocessors. These memories are: faster read-only constant/texture memory and shared memory. The GPU cards are massive parallel devices. They enable thousands of parallel threads to run which are grouped in blocks with shared memory. The blocks are grouped in a grid (Fig. 2).

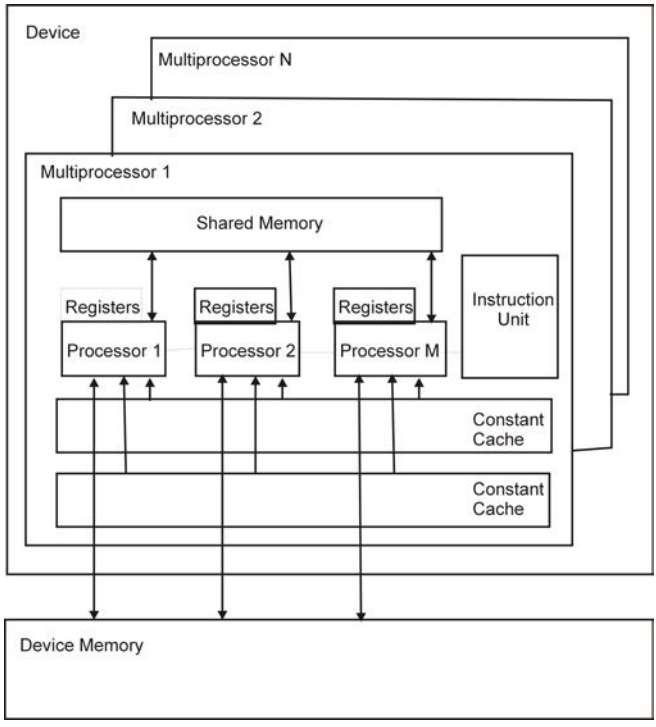

Figure 1. GPU card architecture.

Creating the optimized code is not trivial and a thorough knowledge about the GPUs architecture is needed. The main aspects are the usage of the memories, an efficient dividing code to parallel threads and thread communications. As was mentioned earlier constant/texture and shared memories are the fastest. Therefore, programmers should optimally use them to speedup access to data on which an algorithm operates. Another important thing is to optimize synchronization and the communication of the threads. The synchronization of the threads between the blocks is much slower than in a block. If it is not necessary it should be avoided. The rest of the

architecture aspects will be described in the section about the implementation of the SQL operations.

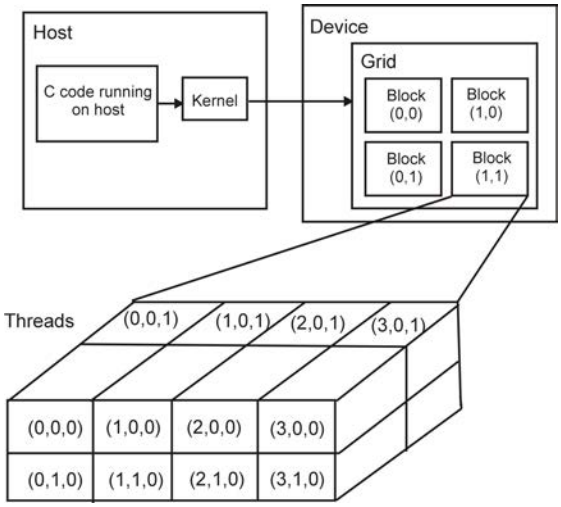

Figure 2. Relation between grid, blocks and threads

3. Implementation of SQL queries

The main purpose of our project is to build an SQL accelerator on a GPU platform. At the beginning we tried a prototype and implemented the most popular SQL operations such as the SELECT WHERE clause and the JOIN operation [13]. Another goal is to build an interface for these SQL operations. This issue demands the creation of a parser which translates instructions from a SQL format to code which can be executed on the GPUs. The parser is built using the ANTLR framework [16]. Next, there are two main possibilities to exploit this parser together with built in SQL operations on the GPU. The first one is to build one's own database system which works only on a GPU. The second solution is to create a plug-in which can be added to the existing database system working on the CPU such as SQLite [14]. It is worth mentioning that an independent system is faster because there is no need to translate the internal format of a commercial database to the representation created for the GPU. Currently, in our project there is a parser which translates the implemented SQL operations to the internal format and sends them to the GPU, where they are executed (Fig. 3).

3.1. SELECT WHERE implementation

There are a few possibilities to implement parallel SELECT WHERE clauses on the GPU. The first method is to divide rows to available threads and execute them on

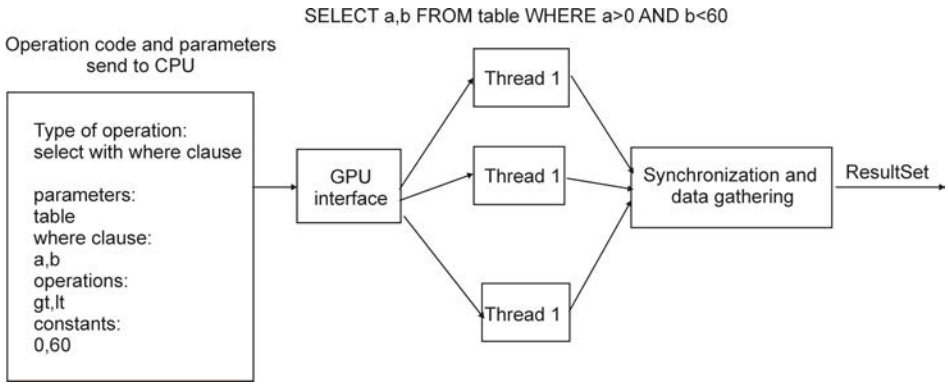

Figure 3. The translation process of an SQL query to the internal format for a GPU.

the SELECT WHERE clause, then synchronize the threads and the result is ready for gathering and sending it to the host. The problem in this case is to allocate enough shared memory for the table on which the operation is done. The fastest case is when the each thread executes a SELECT WHERE on a single row from input table (Fig. 4). In the case when the table has more rows than available threads in one block, it must be divided between blocks (Fig. 5) or threads must execute a SELECT WHERE clause on a few rows. The synchronization between blocks is much slower than between threads inside the block. It should be avoided to not decrease the efficiency of the SELECT WHERE operation. After all computation is done by the threads in all blocks, the data is written to a result table and sent to host.

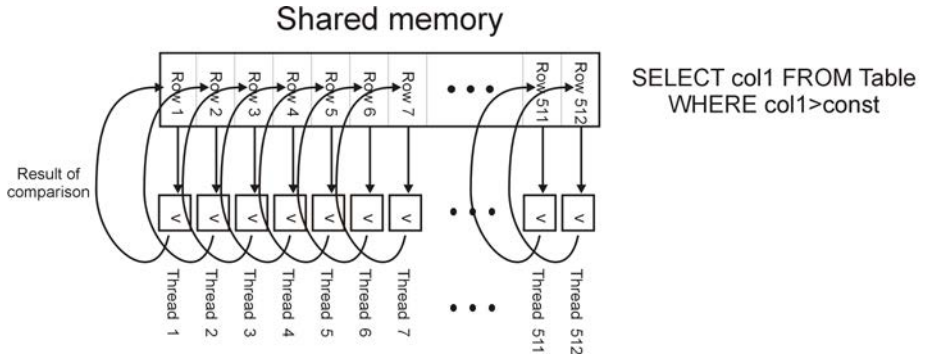

Figure 4. The plan of the SELECT WHERE operation on a single block.

The next possible solution to run SELECT WHERE is using an index prepared earlier. The index is smaller in size than the table so there is a higher probability for it to fit into the shared memory. The index is a table of rows with two values. The first value is the key, the second one is set of pointers to the rows which have a value

equal to the key. In this case each thread checks a single row in the index table and if condition of the select where clause is passed then pointers are written as a result of the operation. In this paper we only concentrate on implementation without index table.

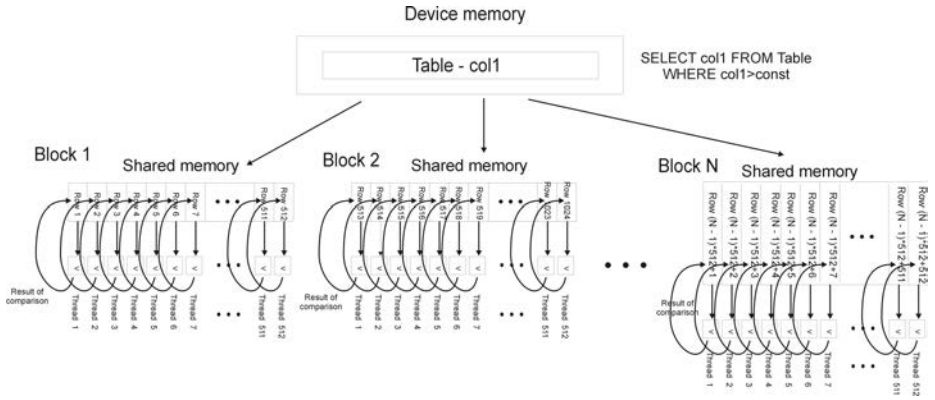

Figure 5. The SELECT WHERE operation divided to blocks.

3.2. SELECT JOIN implementation

The JOIN operation is a Cartesian product of the two tables, in which the result table has rows created by joining these rows in the tables in which the join columns have the same value. This operation is more complex than the SELECT WHERE clause. In our framework the rows in one table (table with more rows) are divided into threads as in the SELECT WHERE clause. Each thread runs the JOIN operation for its rows. It takes each row which it received from the kernel and scans the second table to find rows with columns of the same value (Fig. 6).

The JOIN operation is a little different in parallel implementation to the SELECT WHERE and causes different problems. The main problem is memory usage. The result of the JOIN operation is the Cartesian product. In the worst cases, the size of the memory needed to store the result of such a query is N*M the number of rows, where N is the number of rows of the first table, M is the number of rows of the second table. Without any additional information gathered about the inserted values, the size of the memory for the worst case must be allocated. For example, if N=1000 (every row has 3 integers) and M=500 (every row has 2 integers) in the worst case the result table can have 1000*500 rows with $(2+3)$ size row each. As it is shown the size of the memory needed to store the result increases extremely. Therefore gathering information about the number of every value in a table must be gathered to avoid allocating redundant memory. In our system we monitored it while inserting rows to tables or when transforming data from other databases. As it will be described it enables decreasing redundant data transfer from GPU to host after JOIN execution.

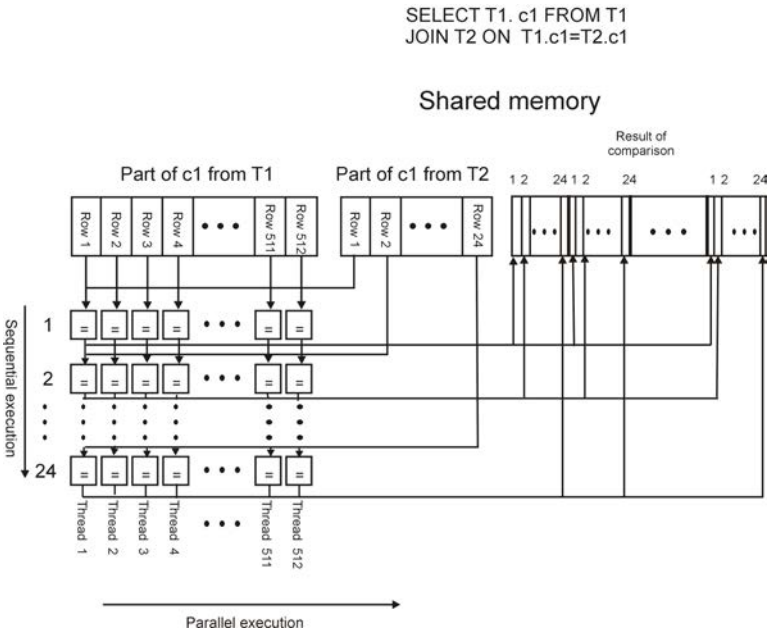

Figure 6. The plan of the SELECT JOIN operation on a single block.

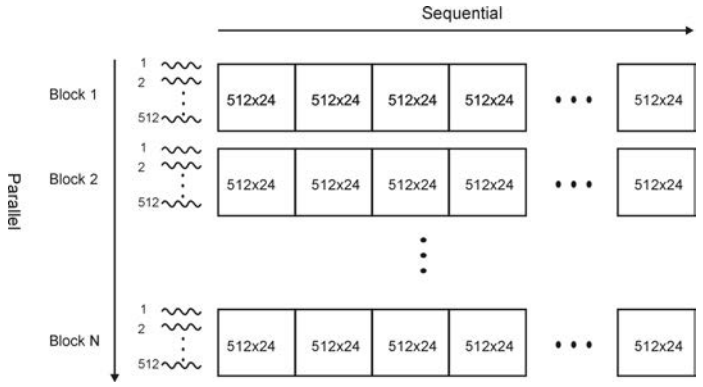

Figure 7. The plan of SELECT JOIN operation divided between blocks.

As described in Figure 7 the tables are divided into blocks to fit their shared memories. Each block has a constant part of the first table (512 rows of joined column). The rest of the free space of shared memory is filled by the part of the second table (24 rows). Each thread in a block knows which parts of the tables are in a block's shared memory. This information is needed to write which rows are joined (Fig. 6). Then after all the computation threads in a block are synchronized and the

indexes of joined rows are sent to the global memory. After all blocks finish their work the result data is sent to host.

4. Results

The results in our research were measured in three cases of the GPU query execution:

- the time of query execution on the GPU,
- the time of query execution with transfer from the GPU to the host (the data was loaded earlier into the GPU's memory before execution),
- the time of query execution with transfer to the GPU and from the GPU to the host.

In our tests the GPU has a transfer speed of about 4–5 GB/s [12]. Table 1 and Table 2 show the results from comparing the time of execution of the SELECT WHERE and SELECT JOIN operations between the GPU and the CPU platforms. The GPU times described in these figures are the time of algorithm execution and data transfer time. As it is seen the SELECT WHERE query in our implementation is much faster on the GPU platform. In the case of the SELECT JOIN we can also observe the speedup. It increases when the size of the input tables rises.

The transfer times of SELECT JOIN operation are measured when any monitoring of compared values is done. Therefore, algorithm must reserve a huge size of device memory to store all possible cartesian products of the query. The results show that SELECT JOIN should be executed with other SQL operations to decrease the size of the result data and transfer time.

Table 1
The results of the SELECT WHERE operation.

Size of table	GPU execution (ms) NVIDIA Tesla m2090	CPU execution (ms) Intel Xeon E5645	Transfer time (ms)
20 480	0.01	2	0.004
0.6 M	0.015	3	0.12
5.2 M	0.1	6	1
20 M	0.31	10	4
90 M	1.27	33	18
340 M	4.61	112	68
640 M	8.63	220	140
1.2 G	18	450	250

It is worth mentioning that the implemented operations were executed without any planning of the operation as it is in the case of the sequential execution on a CPU. It is only necessary to monitor the amount of data needed to send to the GPU and in the case of the JOIN operation only to compute the number of values in the rows.

The reason for this situation is that the GPU runs the SQL instructions in parallel, and very often the sequence of the execution of the internal SQL instructions

is not an important issue. The instructions of the SQL query can run in parallel because they are very often independent. When implementing more complex queries this feature should have the advantage when comparing the time of execution between the GPU and the CPU.

Table 2
The results of the SELECT JOIN operation.

Size of table	GPU execution (ms) NVIDIA Tesla m2090	CPU execution (ms) Intel Xeon E5645	Transfer time (ms)
2000×2000	5.813	21	27.4
4000×2000	5.885	43	39.211
8000×2000	6.093	93	68.380
10000×2000	11.65	114	84

5. Future work

This paper shows a prototype of building an SQL accelerator on a GPU platform. Currently, there are only a few operations implemented. The time of execution of these operations shows that a GPU can be used to accelerate SQL queries. The few drawbacks of the GPU such as the data transfer time and the size of the memory causes the building of a specialized SQL profiler to estimate the time of execution and the chance of a speedup. The next very important issue is to implement complex SQL queries. In this case it is possible to achieve better acceleration because very often more operations can be run independently than in a single SQL operation such as those described in this paper.

Acknowledgements

The work presented in this paper was financed through the research program – Synat.

References

[1] Di Blas A., Kaldeway T.: *Data monster: Why graphics processors will transform database processing.* IEEE Spectrum, September 2009

[2] Bandi N., Sun C., Agrawal D., El Abbadi A.: *Hardware acceleretion in commercial database: a case study of spatial operations.* VLDB 2004, pp. 1021–1032, 2004

[3] Hoff T.: *Scalling postgresql using cuda, May 2009.* `http://highscalablility.com/scaling-postgresql-using-cuda`

[4] He B., Lu M., Yang K., Fang R., Govindaraju N. K., Luo Q., Sander P. V.: *Relational query coprocessing on graphics processors.* ACM Trans. Database Syst., 34(4):1–39, 2009

[5] Govindaraju N. K., Lloyd B., Wang W., Lin M., Manocha D.: *Fast computation of database operations using graphics processors*. ACM SIGGRAPH 2005 Courses, p. 206, New York, NY, 2005. ACM

[6] Ding S., He J., Yan H., Suel T.: Using graphics processors for high performance IR query processing. *Proc. of the 18th international conference on World wide web*, pp. 421–430, New York, NY, USA, 2009. ACM.

[7] Fang R., He B., Lu M., Yang K., Govindaraju N. K., Luo Q., Sander P. V.: GPUQP:query co-processing using graphics processors. In *ACM SIGMOD International Conference on Managament of Data*, pp. 1061–1063, New York, NY, USA, 2007. ACM.

[8] Han T. D., Abdelrahman T. S.: Hicuda: a high-level directive-based language for GPU programming. *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 52–61, New York, NY, USA, 2009. ACM.

[9] Ma W., Agrawal G.: A translation system for enabling data mining applications on gpus. *Proc. of the 23rd international conference on Supercomputing*, pp. 400–409, New York, NY, USA, 2009. ACM.

[10] Che S., Boyer M., Meng J., Tarjan D., Sheaffer J. W., Skadron K.: A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[11] Bakkum P., Skadron K.: *Accelerating SQL Database Operations on a GPU with CUDA*. GPGPU-3, March 14, 2010, Pittsburgh, PA, USA.

[12] `http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf`.

[13] `http://dev.mysql.com/doc/refman/5.6/en/select.html`.

[14] `SQLLite:http://www.sqlite.org`.

[15] `NVIDIACUDA:http://www.nvidia.com`.

[16] `ANTLR:http://www.antlr.org`.

Affiliations

Marcin Pietroń
AGH University of Science and Technology, Academic Supercomputer Center Cyfronet, Krakow, Poland, `pietron@agh.edu.pl`

Paweł Russek
AGH University of Science and Technology, Academic Supercomputer Center Cyfronet, Institute of Electrical Engineering, Krakow, Poland, `russek@agh.edu.pl`

Kazimierz Wiatr
AGH University of Science and Technology, Academic Supercomputer Center Cyfronet, Institute of Electrical Engineering, Krakow, Poland, `wiatr@agh.edu.pl`

Received: 2.11.2012
Revised: 4.01.2013
Accepted: 11.02.2013