Computers and Electrical Engineering 57 (2017) 69–80

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Accelerating relational database operations using both CPU and GPU co-processor☆

Esraa Shehab[b], Alsayed Algergawy[a,b,*], Amany Sarhan[b]

[a] Institute for Computer Science, Friedrich Schiller University of Jena, Jena, Germany
[b] Computer and Control Engineering Department, Faculty of Engineering, Tanta University, Egypt

ARTICLE INFO

Article history:
Received 7 December 2016
Revised 12 December 2016
Accepted 13 December 2016
Available online 20 December 2016

Keywords:
GPU
Query processing
CPU co-processor
CUDA

ABSTRACT

Data is evolving and the number of existing data sources is vastly growing. Therefore, there is a compelling need for effective techniques to store, retrieve and process such massive data. Significant speed-ups at a small cost can be achieved by deploying co-processors such as GPUs. To this end, in this paper, we propose a new hybrid query processing technique that makes use of the capabilities of CPUs and GPUs. The proposed approach breaks down each SQL statement into smaller parts during the parsing process. It then automatically manages the distribution of different query parts to be executed either on the CPU or parallel on the GPU and CPU. To achieve this, we developed and implemented the proposed approach on a SQL server database using the .Net framework instead of working under the Linux environment. The performance of the proposed approach is validated using different workloads and the results demonstrate that the proposed GPU-based query processor achieved speedup up to 39 as fast as multi-core CPUs.

1. Introduction

The growing and evolving data results in amplifying need to develop and employ effective and efficient techniques able to manage such huge amount of data. *Graphics Processing Units (GPUs)* have become emerging and powerful co-processors for many data-intensive application domains including scientific computing and databases [1–4]. This is due to recent GPUs have an order of magnitude higher memory bandwidth and higher Giga floating point operations per second than the multi-core CPUs [5].

Database, most of the time, handles a large amount of data and, accordingly, database management systems (DBMSs) must provide acceptable mechanisms to deal with these extremely amounts of data in an agile manner to provide the results to the user in the least possible time. SQL is an industry-standard generic declarative language used to manipulate and query databases and it is also capable of performing very complex joins and aggregations of multiple data sets [6]. An acceleration of SQL queries would enable programmers to increase the speed of their data processing operations with little or no change to their source codes. In general, the acceleration of databases using GPUs has attracted much attention from both scientific and industry communities. The response time of DBMSs queries using GPUs depends largely on the

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Associate Editor Dr. H. Tian.

* Corresponding author.
E-mail addresses: alsayed.algergawy@uni-jena.de (A. Algergawy), amany_sarhan@f-eng.tanta.edu.eg (A. Sarhan).

http://dx.doi.org/10.1016/j.compeleceng.2016.12.014
0045-7906/

following two factors: (i) *I/O cost*: the time for transferring data from the CPU main memory to the GPU main memory; and (ii) *computational cost*: the time for processing data by DBMSs.

The current research focuses on how to reduce the I/O cost during query processing. In a previous work, the researchers demonstrated that GPU-acceleration cannot achieve significant speedups if the data has to be fetched from the disk, because of the IO bottleneck [7,8]. GPUs can only improve the performance if data is already available in the main memory. A recent research has shown that GPUs can be designed as an accelerator for individual database operations, such as `sort` [2] and `joins` [3]. Another set of approaches implement a database optimizer considering query plans with different operator placements. However, existing approaches and database systems making use of GPU acceleration have a common problem; they need to decide for each database operator, on which heterogeneous processor it should be executed. Typically, each system has a set of analytical cost models, which model the performance behavior of an operator on a particular processor (e.g., CPUs or GPUs).

To deal with this problem, in this paper, we introduce an adaptive query processing approach which executes a SQL optimization plan using a *hybrid query processing* approach. In particular, the approach is to automatically manage the distribution of the query parts to CPUs or/and GPUs. To this end, database queries are first parsed and partitioned into smaller parts, where each part is then directed to be executed either on the CPU only, on the GPU co-processor only, or on both. The main problems arising when deploying both devices are how to partition the query properly; how to automatically decide which part should be executed on which device. That is *the main focus of this reacher is how to automatically decide which part of the query should be executed on CPUs and which others should be executed on the GPU co-processor*.

To sum up, the main contributions of the paper are listed as follows:

- Proposing an adaptive query processing approach that accepts user queries. Each query is parsed and partitioned into parts then the distribution of the query parts to be executed either on the CPU, on the GPU or on both of them is automatically managed.
- Developing a modified query parser based on the basic structures of SQL query parsing.
- Splitting up input query statements into parts then making a decision for dispatching these parts to be executed on a suitable processor rather than using a virtual machine which compiles the SQL statement to opcode model instructions of the virtual machine. These instructions are then executed in parallel on the GPU and rather than individual primitives whereas a primitive is a function implemented directly as an independent CUDA kernel.
- Alternatively executing the `RIGHT OUTER JOIN` query on both CPU and GPU processors by executing the outer loop of `JOIN` on the CPU and the inner loop of `JOIN` on the GPU rather than executing the nested loop of JOIN on the GPU only as in previous research. This greatly reduces the execution time giving a significant speed up of executing this query because executing `JOIN` only on the GPU requires the transfer of large amount of data from/to the GPU that deteriorates the speed up gained by the execution on the GPU.
- Carrying out an extensive set of experiments to validate the performance of the proposed approach comparing it to one of the related recent approaches [9].

The remainder of the paper is structured as follows. Background and a set of the related work are presented in Section 2. In Section 3, we introduce the proposed hybrid query processing approach and discuss its implementation details using CUDA. Section 4 reports the performance evaluation. Finally, in Section 5, we conclude this work and present directions for the future work.

2. Background and related work

In this section, we present the GPU architecture and main GPU-based applications in data management in general, and query processing in particular.

2.1. GPU architecture

Graphics processing units (GPUs) have commonly been developed to employ data parallelism for data intensive computations. GPUs are specialized architectures traditionally designed for gaming applications [10], which can be used as coprocessors for CPUs [4,11]. It is required to transfer data from the CPU main memory to the GPU memory before processing on the GPU. After processing the data, the result has to be transferred back into the CPU memory.

Fig. 1 shows the generic architecture of a modern computer system with a graphics card. The graphics card, called the *device*, that is connected to the *host* system via the PCI Express bus. All data transferred between the *host* and the *device* has to pass through this comparably low-bandwidth bus. Typically, both the host and the device do not share the same address space, meaning that neither the GPU can directly access the main memory nor the CPU can directly access the device memory. A typical GPU, as shown in Fig. 1, consists of multiprocessors, which can be seen as very wide single instruction, multiple data (*SIMD*) processing elements [12]. Every multiprocessor contains a number of thread processors and a shared memory, which is accessible by all thread processors. These units are also called stream processors or thread processors.