Information Retrieval from a Large Database using Query Co-Processing on a Graphics Processing unit

Tushar Prabodhankar¹ Shivam Singh²
¹,²Department of Information Technology
¹,²Management at Golden Gate University, San Francisco, CA

Abstract—Now a day for the retrieving the data from large number of databases is complicated with respect to the time and other measures since we are introduced to new technology such as Big Data where the capacity of storing the information is much higher as compared to other databases. So the retrieval of a particular information is much more complex and time consuming with the help of the traditional processors. The idea behind query processing for the informational retrieval from the huge databases using a Graphics Processing Unit that works on CPU and GPU for in-memory query co-processing. As per the recent research shown they can accelerate more operations with respect to a database, the GPU is capable of performing the tasks with respect to the databases with the help of the derived algorithm such as Sort, Min_Max Join, and CSSSTree etc. Thus we propose to implement information retrieval from a large database using query processing on graphics processing unit.

Key words: GPU, Query Co-processing

I. INTRODUCTION

Today a graphics processing unit (GPU) can be found in various computing devices such as PCs, laptops, gaming consoles and even some cell phones. The main application of a GPU is in the gaming sector on account of it being a massively parallel processor with 10x higher computation and 10x higher memory performance than a CPU. [3] GPU being a magnitude faster than a CPU have found an application not only in the gaming sector but in other fields as well such as information retrieval from large database. Such GPUs are termed as General Purpose GPU (GPGPU). [3] Task such as selection and aggregation queries, spatial joins, and sorting can be performed using a GPU. There are many academic research projects as well as commercial solutions that are based on this approach but none can be considered to be definitive.

Our aim is to develop an in-memory query co-processor using the out-of-order execution capabilities of CPUs and efficient data-parallel processing capabilities of GPUs. In the following, we introduce the preliminaries for GPU and query processing, give an overview of our system, present three GPU-query based algorithms showing the faster compile time that a GPU has in comparison to a CPU and in the process of which we can deal with the new challenges that are posed by this system.

II. PRELIMINARIES

As the GPU is majorly use to run the applications on graphics, the basic data structure in GPU programming is a 2D array, called a texture. An element of a texture, which contains four values, each of which corresponds to a color channel (R, G, B, and A). We are able to access the Textures at the time of rendering passes, where a particular output texture is set as a render target, and multiple textures are used as input. Texture fetching can read data from arbitrary locations in a texture; however, writing to an output texture is mostly limited with respect to the locations. In addition, general purpose programming models allow the database developer to use the GPU without any knowledge of computer graphics. Responsible for managing the various hardware level structuring and databases. It hides the hardware operation details and allows these scarce resources to be efficiently shared.

In this work, to develop algorithms for query processing for general purpose computing framework (NVIDIA CUDA)[2] and graphics API (Microsoft DirectX)[1]. API can utilize specific graphics hardware features, such as the general purpose framework exposes the massively multi-threading parallelism, data scattering capability and fast inter-processor communication available in the hardware.

III. SYSTEM OVERVIEW

In query-processing relational operators like selection, projection, join, and aggregation are used. The implementation alternatives include sorting, hashing, table scan, binary search, and tree index search. Fig1 illustrates system structure of query processing using GPU.

![Fig1: System structure of query processing using GPU](image-url)

The query which is fetched by user to extract the necessary data from the database. The query is supposed to the complex one as we are dealing with the application of complex queries and retrieval of exact data from database. Once it is fetched it gets converted into sub-query and kernal gets allocated to the individual sub query. The sub query is then divided into multiple threads for the parallel execution of the multiple thread which occur simultaneously and it occurs through the global memory which is shared between both the processors. Loading the result back to CPU that is the host part which will display the inquired data from the database.
IV. QUERY CO-PROCESSING

There are mainly three common database operations, namely sorting, tree indexing, and joins. These operations are implemented using GPU based algorithms on both DirectX and CUDA. In the following, the main focus is on both our DirectX implementation as well as briefly discussing the optimizations in CUDA.

A. Using Sort

Our Sort algorithm is based on a bitonic sorting network. A bitonic sequence is a merger of two monotonic sequences. The algorithm proceeds in log2n stages for a sequence of n elements. In each stage i, it performs i steps in total, from step i to step i, to merge two bitonic sequences of size 2i-1 each into a new bitonic sequence of size 2i. In each step, elements are compared in pairs, and the maximum or minimum of a pair is stored. In Sort, we store the array to be sorted in a 2D texture with four color channels. The pixel program compares pairs of elements in parallel, stores the minimum or maximum of each pair locally, and renders the array to another texture. The process continues until the final texture is fully sorted. Sorting using GPU is faster on account of its parallelism. It performs orders of magnitude faster than optimized CPU-based sorting algorithms. Similarly, the bitonic sorting network can be easily mapped to CUDA. Since CUDA exposes fast on-chip memory, which is shared among different processing engines, we can further optimize the algorithm. This optimization is accomplished by using the shared memory to sort all bitonic sequences whose sizes are small enough to fit in it. Due to the small sizes, the number of device memory accesses is reduced, which further reduces the execution time of the sorting operation.

B. Using CSS Tree

The B+-tree index is a pivotal access method for disk-resident databases, and the CSS-tree is a static, in-memory, and cache-sensitive variant. The main feature of a CSS-tree is that the tree nodes are physically aligned as an array without any pointers. Consequently, a search in the tree is performed via address arithmetic instead of pointer chasing. This array-based tree organization is suitable for the GPU; therefore, we implement it for GPUP. A CSS-Tree can be efficiently constructed on the GPU taking a sorted relation as input. Suppose we have a sorted relation of 4N entries. We group them into 4-tuples and store them in a Texel array with the first four entries in the R, G, B, A channels of the first Texel in ascending order, the next four in the second Texel, and so on. Results we obtain from Sort will have automatically assumed this format. We then construct a CSS-Tree with data representing the leaf nodes. The internal nodes, which constitute the directory structure, are computed and stored in a separate array dir. The entire construction process is completed with just one rendering pass. Additionally, the address computation is purely scalar and therefore the four-channel calculation is easily vectored to be processed with a single instruction sequence. While searching for a single key in such a tree offers little opportunity for parallel processing, multiple searches, for example, those in indexed nested-loop joins, fits extremely well in the GPU programming model. The basic idea is to construct a texture for a group of keys to be searched and to perform a rendering pass when going down each level of the tree. Similar to tree construction, the calculation of the array indexes of the children nodes for search can be vectored easily on the GPU. Furthermore, if the group of search keys is sorted, adjacent searches will go through similar paths and visit adjacent or even identical tree nodes, and thus improve the cache hit rate of the texture fetches. An interesting tradeoff is between this cache performance gain and the sorting cost of a group of unsorted search keys. Similarly, with CUDA we can also do the searches one level at a time in order to reduce the latency resulted from random accesses to the device memory. Furthermore, we can store the frequently accessed, higher level tree nodes in shared memory in order to significantly reduce device memory accesses.

C. Using Min-Max Join

Join traditionally, joins can be implemented via nested loops (NLJ), sort-merge (SMJ), or hashing (HJ) on the CPU. In comparison, we propose a new GPU-based algorithm, called Min-Max Join (MMJ), to execute join operations efficiently. This MMJ method utilizes hashing, sorting, as well as GPU-specific features, such as scattering and min-max blending. MMJ assumes the join key, record ID, and other attributes of the two input tables S and T are stored in textures. The output is a texture containing the result of the join operation. The algorithm proceeds as follows: First, we sort the smaller table S by the join key. The resulting sorted table is S’. Next, we compute the range of the positions of S’ records in the texture for each join key value and store these ranges together with their join key values in an auxiliary texture R. This can be accomplished efficiently by using graphics hardware min-max blending, which is exposed to the API. The texture R can be indexed using a hash function on the join key value, and collisions can be detected in a subsequent pass and addressed by rehashing. Next, for each record in the larger table T, we look up R to see if there is a range of records from S’ that joins with this record. If there is a match, we output the potential range to a texture Q. Q is indexed by the record position in T. Next, we cluster the non-zero entries in Q. This can be accomplished with a bitonic sort. Finally, we sum the number of elements on each range in Q to be the number of the join result tuples, determine the position of each result tuple in the final result table, and populate the result table using scattering. This approach runs entirely on the GPU and takes advantage of hardware parallelism in all steps. It works for non-equi-joins by modifying the method used to compute the range in Q. In CUDA, we do not have direct access to hardware min-max blending. However, due to the more flexible framework, we can more easily map traditional joins to the GPU as it can utilize the scatter and inter-process communication provided by CUDA. Especially, we will demonstrate the four traditional relational joins including indexed and non-indexed NLJ, SMJ and HJ on CUDA.

V. PROCESS FLOW DESCRIPTION

The main objectives which we aim to achieve are:

1) The Run time performance of the complex queries using the algorithm like Sort, Min-Max Join etc.
2) The most important advantage using the GPU as a commodity processor that is comparing the compile time of it with CPU. In the following, we describe the processing flow demos, since one of the interesting advantages of programming with graphics processors is that, regardless of the computation being performed, the results are readily available for visualization with the help CUDA.

A. CUDA Based Implementation

CUDA stands for Compute Unified Design Architecture. It acts as an Interface between CPU and GPU. To the CUDA Programmer computer system consist of a host which is traditional CPU and one or more devices which are massively parallel processor equipped with the large number of arithmetic execution unit. The device accelerates the execution of these applications by harvesting a large amount of Data Parallelism. The CUDA program consists of the two or more phases that are executed on either the host that is CPU or the device such as GPU. The phase that exhibit little or no data parallelism are implemented in host code And the data which exhibits rich amount of data parallelism are implemented in device code. The NVIDIA® C compiler separates the two during the compilation [2].

The host is straight an ANSI C code it further compiled by the host’s standard C Compiler whereas the device code is written using ANSI C extended with key words labelling data parallelism function called Kernels. The device code then further compiled by the NVCC and executed on GPU device. In situation where no device is available or the kernel is more appropriately executed on CPU using emulation feature in CUDA (SDK). The Kernel function typically generates a large number of threads to exploits data parallelism.

The Execution starts with host execution (CPU). When a kernel function is launched the execution is moved to devices (GPU) where a large number of threads are generated to take the advantage of abundant range of data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a grid. In CUDA the host and device have separate memory spaces. The reality that devices are typically hardware cards that comes with their own dynamic random access memory (DRAM).

In order to execute a kernel on the device the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to allocated device memory. The CUDA runtime system provides the application programming interface (API) to perform this activity on behalf of the programmer. After the device execution the programmer needs to transfer the result data from the device memory back to the host memory. The host device shows the result. [2]

VI. CONCLUSION

With the rising importance of GPU computing, GPU hardware and software are changing at remarkable pace. This advancement in GPU computing ability is directly reflected on the query co-processing systems since it involves a number of uses by reducing response time, improving the QoS(Quality of Service)efficient management of database to name a few. Thus, we propose query co-processing using graphics processing unit using the high configuration GPU as opposed to the normal CPU increases the computation power as well as enhances speed and performance proving more valuable for future use.

REFERENCES