Data Management Systems on GPUs: Promises and Challenges

Yi-Cheng Tu, Anand Kumar, Di Yu, Ran Rui, and Ryan Wheeler
Department of Computer Science and Engineering,
University of South Florida, Tampa, U.S.A.
ytu@cse.usf.edu, {akumar8, diyu, ranrui, ryanwheeler}@mail.usf.edu

ABSTRACT

The past decade has witnessed the popularity of push-based data management systems, in which the query executor passively receives data from either remote data sources (e.g., sensors) or I/O processes that scan database tables/files from local storage. Unlike traditional relational database management system (RDBMS) architectures that are mostly I/O-bound, push-based database systems often become heavily computation-bound since the data arrival rate could be very high. In this paper, we argue that modern multi-core hardware, especially Graphics Processing Units (GPU), provide the most cost-effective computing platform to catch up with the large amount of data streamed into a push-based database system. Based on that, we will open discussions on how to design and implement a query processing engine for such systems that run on GPUs.

1. INTRODUCTION

Traditionally, RDBMSs follow a *pull-based* design in its query engine: a computational thread is initiated to each individual query the system accepts, and the very data needed are pulled from storage to accomplish execution of that particular query. Recent years have witnessed the increasing popularity of pushed-based DBMS architecture, in which query execution passively depends on a streaming mechanism to feed data to it, and multiple queries are served simultaneously by the same data streams. There are mainly two types of such systems: (1) *data stream management systems* (DSMS) [3, 12] that aim at monitoring and analyzing continuous data inputs from multiple sources (e.g., sensors, stock price updates) in real-time; and (2) *scan-based DBMSs* in which data streams are produced by scanning stored database tables to maximize I/O sharing among concurrent queries [4, 6, 9, 11, 10, 15]. This method has found great success in application domains with read-intensive or read-only workloads such as online analytical processing (OLAP) and scientific data management.

SSDBM '13, July 29 - 31 2013, Baltimore, MD, USA

At runtime, the push-based query processing engine consists of pipelined query execution plans/trees similar to those found in traditional RDBMSs. Any operator (i.e., node) in a query plan can be shared by multiple query pipelines that form a runtime query operator network, with the output of an upstream operator as the input to the next one(s) in the workflow (upper part of Figure 3). One important note is that, with the popularity of various scientific and business applications, support of complex analytical operators is a must in such systems - we can view such operators as user-defined aggregates. Each operator has its own input queue for holding the intermediate results waiting to be processed. In a streaming manner, data tuples are pushed to the operator networks to be consumed.

Push-based DBMSs are proposed with very high data input rate in mind, driven by the needs of typical applications such as large-scale surveillance and scientific data analysis. The push-based design has essentially transformed database systems from I/O-bound to computation-bound. For example, since random reads are minimized, a scan-based DBMS can input data at a rate reaching the maximum bandwidth of the storage system (e.g., a few GB/s in a storage array equipped with FibreChannel connections). As a result, the demand on computational capacity to handle such large volume of data in the query engine exceeds the capacity that can be provided by a few CPU cores found in a typical server. In this paper, we argue that modern GPUs with general programming capabilities fill this gap by offering the most cost-effective computing platform to catch up with the data streams. Therefore, the main purpose of this paper is to initiate discussions on the design and implementation of a query processing engine that run on GPUs for push-based database systems. Particularly, we will provide detailed justifications on using GPUs to implement such a system based on our conceptual reasoning and empirical evaluations in Section 2. We also sketch a design of the query engine and the challenges in system design in Section 3.

Comparison to Related Work. In addition to the ocean of literature about using GPUs in many high-performance computing (HPC) application domains, the database community has also studied query processing on GPUs [2, 13, 7], with much efforts invested into various techniques to improve performance of joins [1, 5]. However, such work falls into the same paradigm found in traditional HPC work: they all focus on processing individual computational tasks (i.e., database operators) using GPUs as co-processors. The research advocated in this paper, on the other hand, is about

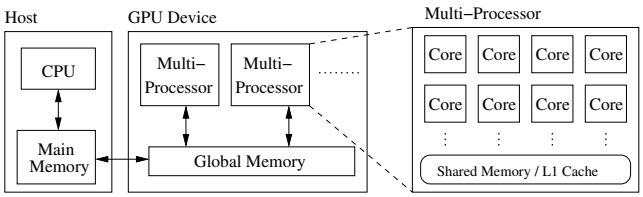


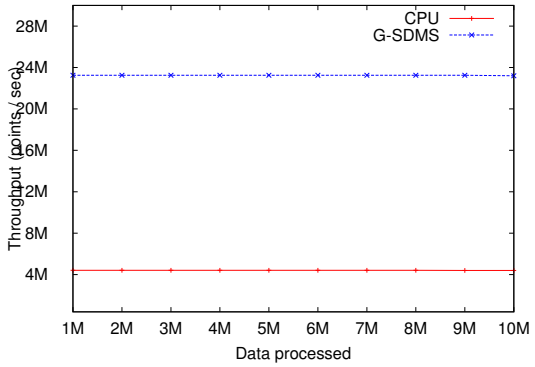
Figure 1: Typical GPU architecture

building a high-throughput *system* software that supports heterogeneous applications (that form a network of concurrent operators) at runtime. Specifically, the primary design goal of such system is to *maximize the utilization of the computational power of the GPU hardware to enable data processing with the highest possible input rate*. The challenges imposed are far more complex than those in implementing and optimizing algorithms for individual database operators.

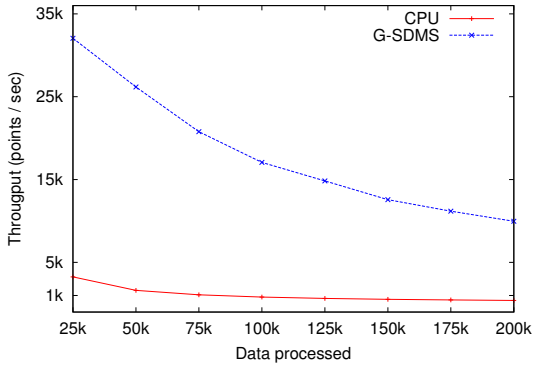
2. GPU ARCHITECTURE AND EXPECTED SYSTEM PERFORMANCE

It is worth highlighting the architectural features of GPUs before we touch the system construction issues. Modern GPUs have evolved into highly parallel general-purpose computing platforms with tremendous computational power and high memory bandwidth. As shown in Figure 1, a typical GPU architecture consists of the following components: (1) *Processor cores* that can execute multiple threads in a Single-Instruction-Multi-Data (SIMD) fashion. The threads can be grouped into *blocks* and each block runs on a processor core. Core binding cannot be done by the software – hardware can freely schedule a block to any physical core on the device. (2) *Multi-level memory.* At the GPU card level, all threads can access the *global memory.* There is up to 6GB of global memory in one GPU card. For each group of cores (such a group is named a *multiprocessor*) there is *shared memory* visible to all the threads of the block. The shared memory, although of a small size (e.g., 48KB), has much higher bandwidth and lower latency than the global memory. Note that both the global and shared memory are directly addressable by user programs. (3) *Communication:* There is no explicit mechanism such as signaling for inter-thread communication.

Conceptually, the performance boost of using GPUs for database systems can be seen at two levels. First, with aggregated computing power of the large number of cores, a GPU can be used to process computation-intensive analytical queries with a fraction of the processing time and energy required by CPUs. Second, with the high bandwidth of its on-board memory, GPUs are also appropriate for running data-intensive queries. For example, the GeForce GTX 570 has global memory bandwidth of 150GB/sec – about 10 times that of a typical DDR3 module. Such numbers can be regarded as a lower bound of the performance improvement one may expect from a GPU-based implementation of the push-based DBMS. While research done on different HPC tasks reported speedup of up to thousands of times [8], others [14] believe the actual performance boost will be at the 10X level. In databases, relational joins are found to be up to 7X faster when implemented in GPUs [1]. However, new generations of GPUs have many times of computational capability than those used in [1] and [14] thus we believe the



(a) Heavy kernel excuded



(b) Heavy kernel included

Figure 2: Throughput of the GPU and CPU versions of a query network running different types of kernels

actual performance gain will be much more promising. One problem in GPU computing is the overhead of transferring data from main memory to the GPU memory. Luckily, in the push-based database system, such overhead can be effectively reduced by the *stream processing* mechanisms offered by recent GPU devices. Specifically, a GPU *stream* is a runtime entity that interleaves data transferring and computation. Up to 16 different GPU streams can run simultaneously with each stream encapsulating its own set of kernels, thus providing an environment of concurrent processing of heterogeneous tasks. In our system with multiple tasks running at the same time, one stream can read data while another performs computation using the cores.

To verify the above ideas, we report the performance of a proof-of-concept system we built (see Section 3 for details of system design). We implemented a query network and executed it on an NVIDIA GTX 570 device installed on a host machine with an Intel i7 4-core CPU running Ubuntu 10.04 and CUDA (compute unified device architecture) 4.0. The query network consists of 20 operators, among which four were selection operators, three were analytical operators taken from scientific domains, and the rest were combinations of the two types. Each of these operators is implemented as a function called *kernel* in the GPU program and launched with 1024 threads in a separate CUDA stream. Data is fed to these operators in chunks from a host-memory-based synthetic data generator. Another similar network was implemented in CPU for comparison. The throughput

measured in both systems is shown in Figure 2(a). The throughput is stable for both GPU and CPU networks in which each operator consumes almost equal amount of data. The GPU throughput is up to 5 times higher than the CPU. The network can process up to 549 MB/s of data. Since this is still far from the maximal bandwidth of a typical data storage system, we can clearly see the demand of computing power in push-based systems. Kernels in the above experiments have constant processing time for each tuple. We also tested a different network by introducing an extra kernel (i.e., computing 2-body correlation functions for spatial data) that carries a very heavy processing cost and the unit cost increases with data size. As seen in Figure 2(b), the data processing throughput dropped from 24M/s to around 25K/s, and the GPU-based system delivers 9-24 times higher throughput. The overhead of the network increases significantly with the amount of data input. Note the CPU results are for optimized multi-thread implementations of kernels on all 4 cores. The advantages of GPU over CPU for data processing is evident from these results, with more significant improvements demonstrated when the data processing tasks become more computation-intensive.

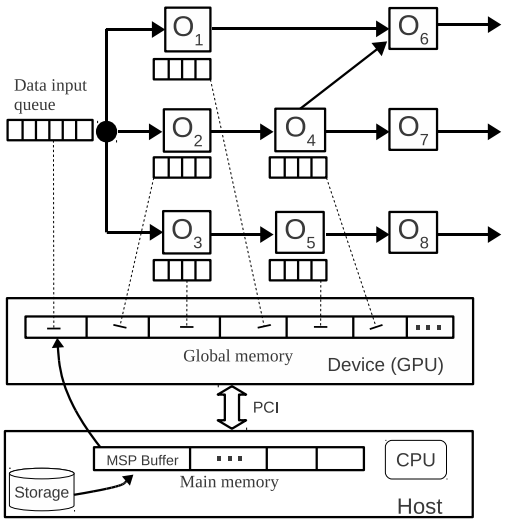


Figure 3: Mapping between a query network to the GPU-based query engine runtime environment

3. SYSTEM DESIGN AND ISSUES

In this section, we sketch the basic design rules of the query engine based on our understanding of the GPU architecture and the CUDA environment. At runtime (Figure 3), it is launched as a GPU process and executed in multiple threads. The CPU will serve as the central control of all GPU kernel processes. The global memory is used as data buffer for the GPU kernels. Each operator will deposit its output data to its own output queue – no input queue is needed. Each queue will be assigned a chunk of global memory. Note that each downstream operator needs to keep the position of unprocessed data in the queue, as one output queue can be shared by multiple operators (e.g., O_4 in Figure 3). To run operators concurrently, each kernel is encapsulated into a GPU stream - one stream can hold more than one kernel.

At runtime, data (from either a remote source or a table scan) first enters a buffer in the host memory (e.g., the 'MSP buffer' in Figure 3). From there, data is transferred by chunks to fill its corresponding GPU (global) memory buffer as the first step of GPU stream processing. After the buffer is filled, the operator kernels are spawned to process the data. A buffer can be implemented as a circular array such that the operators can continuously run as long as there are new data. When a buffer is depleted, we can either keep the (data consumer) kernel wait or terminate the kernel and relaunch it at a later time. We prefer the second solution in GPUs because: (1) current GPU programming environments only support busy wait; and (2) context switch in GPUs carries much smaller overhead than that in CPUs.

3.1 Issues

It imposes many challenges to optimize the above system design towards high data processing throughput. Such challenges, being very different from those we faced in building CPU-based database systems, call for much efforts from the database research community.

Operator Scheduling: In CPU-based data stream management systems, a scheduler is used to explicitly determine the processing order of individual operators. Such a scheduler is not preferred in GPUs since the binding of a kernel process to the available GPU cores cannot be done explicitly by the software. Instead, we enjoy concurrent processing of multiple operators via the GPU stream processing mechanism and leave scheduling to the hardware. An immediate problem is *how to pack operators into GPU streams?* On one end of the spectrum, we can pack the kernels of many operators into one stream. The problem is: kernels in a GPU stream will be processed in a sequential rather than pipelined manner (as opposed to what we might expect even for chains such as $O_3O_5O_8$ in Figure 3). As a result, intermediate buffers can be overflown since it is difficult to predict the amount of data generated at each node. Therefore, we favor the idea of packing very few kernels (or even one kernel) into a GPU stream. As streams can run concurrently, we can simulate the ideal situation of a workflow system: each operator makes progress at the same time.

Kernel Re-compilation: Following the reasoning of operator scheduling, we can implement different operators in a single GPU kernel. For example, a single kernel that encapsulates the operator chain $O_3O_5O_8$ in Figure 3 can process the data in a real pipelined manner such that intermediate buffer size can be minimized. More importantly, due to the reduced size, we could even put the buffer in the shared memory for extremely fast processing. However, such aggressive optimizations may require recompiling the kernel(s), since the query network is dynamic (as queries come and go). Another difficulty is to determine the level of operator merging: putting more operators in a kernel does not necessarily mean better performance – the increase of the number of conditional branches in a kernel is known to degrade throughput. Such observations bring very interesting issues in code management and cost/benefit modeling.

Resource Allocation: Unlike traditional HPC systems, in the push-based system, we need to dynamically adjust the resource distributed to concurrent kernels such that throughput (i.e., overall data consumption rate) is maximized. In-

tuitively, a kernel with a low (high) data consumption rate should be assigned more (less) resources. First, we have to identify the software knobs to implement such decisions, i.e., how to increase the number of GPU cores used for one kernel by 10%? We have thoroughly explored this and are confident that it can be done via changing the *number of blocks* and *threads per block* – these are the two parameters one has to set in launching a kernel. The more difficult part is obviously to compute the amount of resources each kernel should occupy at runtime. From our experience, such a (optimization) problem can be mapped to a feedback control loop, which requires non-trivial system modeling and controller design.

Data Input Control: Even with proper resource allocation, the query engine may still fall short in computing capacity to handle the data inputs. If data comes from a table scan, we can slow down the data generation by reducing the I/O bandwidth used. For data streams, mechanisms such as load shedding and data triage have to be deployed. In either case, the level of decrease of data input rate needs to be carefully studied. Again, this calls for a series of modeling and control efforts.

Operator Behavior Modeling: In addition to the computational steps to obtain the results, a kernel in the proposed system has to be implemented with interfaces for the system to conduct behavior modeling. Such modeling is essential in various aspects of system optimization (e.g., kernel recompilation and resource allocation) and can be done offline by running different workloads on the same GPU hardware. For that purpose, the suite of performance counters (i.e., the numbers of low-level hardware operations such as global memory accesses) provided by CUDA can be used to generate a robust profile of the resource consumption patterns of the tested kernels.

Scaling Up to More Devices: One advantage of GPU-based systems is that multiple (up to 8 on a single motherboard) cards can be installed in a server to reach very high capacity. Several problems mentioned above face more difficulties in dealing with the multi-card scenario. Basically, with the number of resource units increasing, some of the optimization problems become more difficult. Furthermore, the communication among different cards via the PCI-E bus generates a new bottleneck and translates into an extra constraint in such problems.

Heterogeneous Devices: Another line of thinking is to design hybrid systems in which multiple types of hardware systems are used. Co-existence of such hardware (e.g., CPU, GPU, FPGA, ...) in the same node or even the same chip is expected to be a normal scenario, as evidenced by ARM processors that integrate CPU and GPU cores and unified software platforms such as OpenCL. System design goal under such scenarios will still be to *maximize the overall utilization of all resources*. Resource allocation is again the key problem. Apparently, the different behavior of different hardware systems (even in processing the same task) adds an extra dimension of complexity in achieving that goal. We envision a heuristic-based strategy to handle such diversity: we start from an initial configuration based on our understanding of the behavior of existing systems that only deal with one type of processor (e.g., CPU). We then incrementally locate better plans by conduct a series of "what-if" checks on modified versions of the current plan.

4. CONCLUSIONS

This paper advocates investing research efforts into push-based database systems on GPUs. To handle high-bandwidth data inputs, push-based database systems typically face large demand on computing capacity that cannot be easily provided by CPUs. Due to the unique architecture of modern GPUs, the design, implementation, and runtime optimization of such systems impose interesting challenges. We present an initial design of the proposed system, an empirical evaluation of the design, and discuss the most relevant research thrusts related to this topic.

5. REFERENCES

[1] B. He *et al.* Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[2] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the Workshop on GPGPU*, pages 94–103, 2010.

[3] D. J. Abadi *et al.* Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[4] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[5] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *DaMoN*, pages 55–62, New York, NY, USA, 2012. ACM.

[6] M. Zukowski *et al.* Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, pages 723–734, 2007.

[7] N. K. Govindaraju *et al.* Fast computation of database operations using graphics processors. In *SIGMOD*, pages 215–226, 2004.

[8] NVIDIA. CUDA Showcase. . `http://www.nvidia.com/object/cuda-apps-flash-new.html\#`.

[9] P. Unterbrunner *et al.* Predictable performance for unpredictable workloads. *Proceedings of VLDB Endowment*, 2:706–717, 2009.

[10] P. W. Frey *et al.* Spinning relations: high-speed networks for distributed join processing. In *Proceedings of DaMoN*, pages 27–33, 2009.

[11] S. Arumugam *et al.* The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530, 2010.

[12] S. Chandrasekaran *et al.* TelegraphCQ: Continuous data flow processing for an uncertain world. In *Proceedings of the CIDR*, 2003.

[13] E. A. Sitaridi and K. A. Ross. Ameliorating memory contention of OLAP operators on GPU processors. In *DaMoN*, pages 39–47, New York, NY, USA, 2012.

[14] V. Lee *et al.* Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, pages 451–460, 2010.

[15] V. Raman *et al.* Constant-time query processing. In *ICDE*, pages 60–69, 2008.

Acknowledgments

Yi-Cheng Tu is supported by an award (IIS-1253980) from the National Science Foundation (NSF) of USA.