KIWI Project@Data Management Research Section
ETRI Technical Report 15ZS1410-TR-43

218 Gajeong-ro, Yuseong-gu
Daejeon, 305-700, South Korea
http://www.etri.re.kr/
http://sungsoo.github.com/

MapD

Massively Parallel Database

Sung-Soo Kim
sungsoo@etri.re.kr

요 약

In recent years, there has been a focus on using distributed clusters of computers to compute aggregates and other statistics for massive datasets. While these approaches, such as the popular MapReduce framework, are invaluable for extracting actionable information from petabyte-sized datasets, such methods typically require large amounts of hardware (even on a per-byte basis) and suffer from relatively high computational overheads due to the large amount of inter-node communication and synchronization involved. Moreover, for medium-to-large size datasets of less than a few hundred gigabytes, such methods introduce latencies that prevent real-time analysis of data, something immensely valuable in diverse fields such as business intelligence, disaster response, and financial market analysis, among others.

1 Introduction

The system described within, *Massively Parallel Database* (MapD), takes advantage of the immense computational power available in commodity-level, of-the-shelf GPUs, originally designed to accelerate the drawing of 3D graphics to a computer screen, to form the backbone of a data processing and visualization engine that marries the data processing and querying features of a traditional RDBMS with advanced analytic and visualization features. As will be shown, MapD is a unique general-purpose modular system that allows for real-time querying, analysis and visualization of "big data" in real-time on hardware ranging from sub-USD1,000 commodity laptops and desktops all the way to High Performance Computing (HPC) clusters with hundreds or thousands of nodes. Moreover, since MapD provides a vertically-integrated end-to-end solution for data querying, visualization and analysis – it avoids latencies that would otherwise be incurred formatting data and needlessly moving it in and out of ultra-fast GPU memory across the relatively slow PCI bus, which would be required if several different (GPU- based) programs were used to perform the same functions, each with their own protocols.

2 Technical Background

MapD is a software system that is designed to run on a hybrid architecture of GPUs and CPUs. Every modern computer has at least one CPU, and increasingly computer CPUs are divided into more than one sub-processing unit, called cores. It is not uncommon, as of early 2013, for a high-end laptop or desktop computer to have four cores, and many servers have up to 16 cores. The trend towards more cores has been propelled by the inability of computer chip designers to continue to increase chip clock speeds (i.e. the number of computational cycles a processor can complete per second), as the amount of energy required and heat produced increases super-linearly above a certain point, rendering further increases in clock speeds impractical. To counter this "clock-speed wall", processor designers have resorted to other means to continue the inexorable exponential increase in processor speeds that both home and business consumers have come to expect. One strategy has been to increase the complexity of chips, improving the chips' capacity for branch prediction (which allows data to be pre-

fetched from memory before execution) as well as deepening the processor's pipeline, effectively increasing its throughput. The other strategy, as hinted at above, has been to increase the number of processing units in a physical chip. Theoretically, twice the number of processors allows for twice the number of instructions to be processed at once and thus provides twice the computational throughput. However, increasing the number of processors introduces a number of complications. First, some algorithms, such as compiling computer code within the same compilation unit, are very difficult to parallelize. Second, even the use of multi-core architectures to accelerate "embarrassingly parallel" architectures typically requires rewriting major parts of an existing codebase to take advantage of the extra cores. In between these two extremes lies the majority of algorithms, those that cannot be trivially parallelized, but with the right (likely non-obvious) algorithm, can still enjoy significant speedups when run in parallel. Such cases typically require even more extensive rewriting of existing code, as the algorithms used to process data may now be completely different than those used in the serial case. GPUs can be seen as the extreme embodiment of the second approach to overcoming the clock-speed dilemma. The modern GPU evolved in a race to provide the most cost-effective way to perform the massive amounts of computation necessary to render 3D graphics for video and console games – which typically involve hundreds of millions or billions of identical matrix operations per frame to compute the correct projection of vertices in 3D space onto a 2D screen and then fill the resulting polygons using a lighting-dependent shading algorithm.. Hence, emphasis was placed on endowing graphics cards with a large number of architecturally simple cores running at relatively slow clockspeeds, at least relative to the high-frequency, deep- pipelined, heavily-cached world of CPU cores. Moreover, since every core was designed to execute and perform the exact same operation in lockstep with every other core, GPUs ended up being built with a relatively low amount of control architecture, the part of the chip that enables the divergent execution of different branches of code. As such, the first programmable GPU was not offered until 2011, and even modern GPUs do not allow each core to execute independently (Nvidia's architecture only allows for control at a resolution of 16 threads (processors), called a half-warp). GPUs compensate for these limitations by using a massive amount of cores. To date, state of the art consumer/gamer GPUs such as the Nvidia Titan might have 2,668 cores and be capable of 5 Teraflops of single precision performance and 1.48 teraflops of double precision performance. Two such cards installed together into a standard desktop computer would be faster than the fastest supercomputer in 2000, the ASCI RED, that took up 1600 square feet of space, used 850 KW of electricity, and cost USD55 million to build. In contrast, the two Titan GPUs described above cost approximately USD1,000 each, use a combined 500 watts of electricity at peak load, and can be easily be housed in a desktop computer system. The disparity between computational speeds over the last decade is due to the disruptive parallel architecture of the GPU. For instance, the performance of modern multicore CPU chips, offered by leading manufacturers such as Intel and AMD, is measured in tens of gigaflops, approximately a factor of 100 times slower compared to modern GPUs at a similar price point. The scientific community has in particular taken note of the computational potential of GPUs, and currently many workloads such as protein folding and physics simulations are accelerated using GPUs. However, most of these programs are written as single-use scripts or programs and are not applicable to problems outside of the specific problem domain for which they were designed. Cross-purpose data management software platforms that use CPUs and - GPUs (as well as other massively parallel cards) in tandem to both perform data warehousing and analytic tasks have been slow to develop because: 1 The parallelization process for most algorithms is difficult, particularly in the case of a GPU, which has less tolerance for divergent execution than a multi-core CPU; 2 Contemporary programming languages and paradigms for GPUs are relatively under utilised and do not integrate well with existing codebases. For instance, writing code to run on GPUs requires a working knowledge of un-orthodox coding languages (in the sense that such languages are not typically taught in computer science programs), such as CUDA for Nvidia's cards or The Kronos Group's OpenCL; 3 The database architecture most commonly used in conjunction with discrete GPUs requires the GPU to communicate with a CPU and transmit memory over the Peripheral Component Interface (PCI) bus, which under the current standard (PCI 3.0) can only transfer data at approximately 12 GB per second. Resultantly, by using MapD in conjunction with one or multiple GPUs or CPUs, data can be processed faster than in the case described above where computational speed is limited by the speed that

data can be transferred to and from the CPU. 4 Memory on GPUs is often limited compared to that found on the CPU. For example, state of the art GPU cards may have 6GB of GDDR5 RAM, whereas a high-performance server might have 128GB or even 256GB DDR3 RAM. The relatively limited amount of memory available on current GPUs is currently a serious constraint for GPU programmers. 5 The majority of contemporary research focuses on the prevailing paradigm employed to process "big data", Map-Reduce and cluster computing. These take advantage of the scalability and high thoroughput available when using massive amounts of hardware. Alternatively, MapD and a GPU-centric database architecture focuses on the ultra-low latency that can be achieved when processing big data using one or more GPUs. As described below, MapD uses many innovative techniques to overcome these limitations. DESCRIPTION MapD uses a hybrid multi-CPU/multi-GPU architecture running across multiple nodes. This architecture allows for the massive parallelization of the querying, analysis, and visualization of big datasets, and results in an increased speed of processing, in the order of multiple orders of magnitude, across many of the various big data workloads common today. The MapD architecture works in conjunction with deep support for GPU-generated visualizations and allows for the practically real-time exploration of many large datasets by multiple clients in ways that were formerly impossible. MapD utilizes a set of architectural, procedural and programming innovations to contemporary paradigms of big data analysis and computation to achieve ultra-low latency in the processing of big data workloads. Processing times are generally on the order of milliseconds as opposed to seconds, minutes or hours using other contemporary big database paradigms. The innovative features that are incorporated into MapD are, but not limited to: 1. MapD is a GPU data analytical system that systematically exploits both multi-GPU architectures and multi-node architectures. Its design mitigates the costs associated with partitioning data both over multiple GPUs and multiple nodes. 2. MapD uses a three-layer memory hierarchy where GPU memory acts as ultra-fast buffer pool atop of a CPU buffer pool, which in turn sits atop data stored on hard disk. 3. MapD returns indexes or compressed bitmaps of matching rows from the GPU to the CPU after a filter operation rather than the post-filtered projected rows themselves. MapD thus overcomes the large GPU-to-CPU bandwidth costs associated with transferring these rows over the relatively slow PCI bus. 4. GPU query functions are fused as much as possible to minimize memory access costs during GPU-assisted evaluation of queries. Previously GPU database designers have resorted to fusing kernels by hand (i.e. combining the '¿', 'AND', and '¡' operators into one function if this is a common workload for the database). In contrast, the first time MapD sees a certain type of query, it actually generates GPU assembly code for a given query plan and then compiles it to allow for the generation of arbitrarily complex fused kernels for queries that could not be anticipated at compile time. After this initial compilation, these compiled kernels are then stored by the system in a dictionary for future re-use so that the up-front cost of query plan compilation is not incurred again. Furthermore, compiled kernels are stored in file, only being deleted if they are not used again within a user-defined number of queries (the user can choose to never delete compiled kernels if disk and memory space is not an issue). 5. Query optimization for complex queries often involves testing a huge number of possible query plans. Query optimizers found in modern databases typically "give up" after a predetermined amount of time, or constrain the search space to a limited subset of all possible query plans. Both techniques often result in sub-optimal query execution times. To address this issue, MapD experimentally uses the computational power of one or multiple GPUs to determine the optimal query plan for a given query, significantly increasing the number of query plans that can be run-time estimated in a given amount of time, thus increasing the likelihood that the most effective query is found, in turn improving overall system performance. Furthermore, MapD uses its built-in high-performance histogramming capabilities (see 6 below) to improve the statistics that the query optimizer uses (i.e. it improves selectivity optimization) 6. The creation of an SQL extension along with supporting GPU query architecture to systematically handle the binning (histogramming) of data along an arbitrary number of continuously-valued dimensions, afterwards applying any SQL-standard or user-provided aggregate function, and then allowing for the post-processing of this binned aggregated data using arbitrary (including user-defined) convolution filters. This allows the database client to request a heatmap of the percentage of data points in a certain area matching a given query in the same way as it allows for the generation of a time graph of minimum, maximum and average temperature for a given map extent. Convolution filters can

then be used to perform functions such as spatial aggregation (via gaussian blur or box blur) or finding local extrema in the data. Finally, MapD allows for the systematic export of this post- binned/aggregated/convoluted data into a variety of formats, including but not limited to ordinary tabular format, CSV, JSON (useful when serving as a web server), PNGs (returned for Web Mapping Service requests), and finally as the basis for OpenGL rendering which can either be sent straight to the user's screen if the MapD server is actually on the same node as the client, or out over a TCP connection via compressed H.264 video (where compression is performed on the GPU). 7. An algorithm to conduct hyper-fast spatial joins (for example, finding the number of tweets in Texas when the tweets only contain coordinate information). This is done by using the GPU to rasterize in parallel a polygon vector file (such as a shapefile), and then pyramiding it into a lookup tree. Finding the polygon that contains a given point then becomes as simple as looking up the correct pixel in the raster tree and then returning the polygon ID for that pixel. Edge pixels are recorded as such and require an actual intersection test using the underlying vector geometries, but these cases are rare enough so as to have a negligible impact on the overall speed of the algorithm. Preliminary tests have shown speedups of over 1,000,000 times compared to in-memory implementations in PostGIS (a PostgresSQL extension), partly a function of the increased speeds of the GPUs (which are exceptionally fast at texture lookups), but mostly due to the design of the algorithm in question. 8. A novel algorithm to store the content of text data using the products of prime numbers, leading to both computational speedups and significant data compression. This is called "prime encoding". 9. MapD allows for SQL query results computed using its database architecture to be seamlessly passed to various processing modules (such as an included graph analytic package). For example, a user can filter a dataset to filter only those records pertaining to Facebook users from California, and then a compute a real-time 3D force diagram on the results and associated link records, finally passing the data into the provided OpenGL rendering module. One big advantage of MapD's vertically-integrated architecture as an end-to-end data querying, processing, analytic and visualization engine is that when possible, data never needs to leave GPU memory as it is manipulated by successive (and perhaps fused) kernels. This leads to huge speedups over competing solutions that are more piecemeal, even those that run part of their workloads on GPUs. 10. A novel algorithm for performing deep geo-location on documents using machine learning algorithms on the GPU. "Deep geolocation" is here defined to be a means at getting a best estimate of the location of a document's author - where location here is a time-weighted function of their past locations as well as their present position (a user that says "ain't" might be from the Southern United States - but the fact that they repeatedly say "Eiffel Tower" and "Louvre" may suggest that they are currently in Paris). Previous geo-location systems have only relied on pre-generated dictionaries or gazateers containing place names, ignoring various language features that might provide a clue to the user's location (for example, the use of "wicked" near Boston, the use of "bra" instead of "bro" near Atlanta, etc.) The GPU processes a training dataset (such as a dataset of tweets and their geolocations), and then for any document, uses a spatial (2D) Bayesian classifier or other suitable machine learning algorithm to determine probabilities of a user belonging to a given set of N equally sized cell defined by an X and Y extent (a good choice might be a 4X4 grid covering the area in question) The algorithm recursively descends into a given cell if the probability of the document being produced in that cell was higher than a certain amount, stopping when the probability of all sub-bins of a given cell are approximately the same, suggesting that the algorithm could not obtain any more resolution on the document's location (for example, some users may only be able to be located to the southwest). For other users however, the algorithm may not only be able to determine with high probability that not only are they from New York but also that they frequent a certain bar. The output of the algorithm can be either a probability kernel density heatmap (or pyramid of maps) or explicit probabilities that a user is from a given area represented as a polygon on the map (say a state or census district). 11. A novel algorithm to determine "trends" in a dataset over any arbitrary extent in time or space. This is currently referred to as "deep trend detection". Given a weighting function that weights uniqueness in time vs. space, the algorithm essentially uses the computational power available though GPUs to create a 4-D histogram (x, y, time, keyword) of percentage matches for the N most common words in teh dataset as well as the M most common bigrams for a given time and geographic extent. While the top-N unigrams are easy to determine over a given sample, the huge Cartesian space required to represent

the space of all possible bigrams (NxN) requires the use of an on-GPU hashing algorithm to keep count of the top bigrams in the limited GPU memory space. It then uses t-tests to test for statistically significant increases in data points for a given term or phrase for a given cell relative to previous time bins for the same cell. The result is then put through a convolution filter to detect geographic blobs - i.e. sets of neighboring cells that all experienced an increase in relative frequency relative to past time bins. If all cells in a given spatial extent experience a similar increase - this is tagged as a trend that is uniform across geographic space. Trends that are confined to a certain group of cells, however, can be plotted on a map. While there are existing technologies to detect geographically-based trends on sources, such as the social network Twitter, due to processing and algorithmic limitations these are limited to only detecting trends in the current time frame and not on-the-fly and in with near-zero latency across time and space. 12. The integration of a graph analytic module that takes filtered results, upon which various graph statistics can be computed (including but not limited to finding the shortest path length between nodes, the average path length between all nodes, and the centrality of a given node). The module also supports the modeling and rendering of iterative force diagrams of graph data. The results can be rendered in real-time using OpenGL or sent over a network connection as a list of positions or a pre-rendered JPG or PNG. 13. Fast GPU-accelerated clustering and k-means functionality that can be applied to any result set. TECHNICAL DETAILS There are a number of features used in MapD that have been implemented, to varying degrees, in existing RDBMS and data analytics packages. Occasionally, these have also been implemented as custom programs as part of larger big data paradigms such as the popular Map Reduce Framework. However, MapD, through a number of innovative architectural, procedural and programming features working in unison with existing paradigms of big data and GPU based database computation, creates a novel hybrid multi-CPU/multi-GPU database architecture for the computation of big data. The base of the MapD system is a hybrid CPU/GPU column-store relational database. A column-store database is a relatively new variant of a traditional RDBMS that stores its data in columns rather than in rows as in a traditional database. This architecture allows for both the more efficient processing of data on the GPU or CPU, or multiple units of both, as well as for optimal use of relatively limited GPU memory. GPU memory is optimized in that the columns, which are frequently used to filter on or aggregate on, can be cached in the memory of the GPU, while the data columns that are not frequently used can be stored in the more-abundant memory located on the CPU. Contemporary database paradigms are built around the concept of a buffer pool – an area reserved in CPU memory to cache certain blocks of records read from disk. The reason for this is that it is possible to read data from CPU memory orders of magnitude faster than to read it from a hard disk. However, since CPU memory is often smaller by an order of magnitude or more than the space available on a hard disk, it is important to design a database architecture that carefully and effectively selects which blocks of data or records are to be cached in CPU memory. Much research has been conducted on this subject, with various strategies developed to determine which data or records should be cached , or alternatively, which records should be the first to be replaced (via a replacement strategy) when new memory is read from disk to main memory. MapD is designed with this basic database architecture in mind. However, MapD does not use a dual-level memory model, where data moves between CPU memory and memory on a hard disk. Rather, the MapD architecture implements a three-level model of memory and data synchronisation. This three-level model is arranged in a pyramid, where each successive level of memory is slower computationally but larger in size than the last. The fastest and smallest level is the memory on the GPU, next is the memory on the CPU and, finally, the base of the pyramid, being the slowest but most plentiful, is the memory on the hard disk. Every level of MapD's memory hierarchy is a mirrored subset of the level below it. For example, if a given dataset is one terabyte in size, 80 gigabytes of the most frequently used data or records might be mirrored and stored on the CPU memory. Then perhaps the most frequently used 24 gigabytes of those 80 gigabytes will also be mirrored and stored in GPU memory (given perhaps 4 GPUs with 6GB of memory each). See Figure 1 for an overview of MapD's architecture.. Figure 1

참고 문헌