An In-GPU-Memory Column-Oriented Database for Processing Analytical Workloads

Pedram Ghodsnia
Supervised by Edward P.F. Chan
University of Waterloo
200 University Ave. West
Waterloo Ontario Canada
+1 (519) 888-4567 x33263
pghodsnia@uwaterloo.ca

ABSTRACT

Due to ever increasing demand for fast processing of large analytical workloads, main memory column-oriented databases have attracted a lot of attention in recent years. In-memory databases eliminate the disk I/O barrier by storing the data in memory. In addition, they utilize a column-oriented data layout to offer a multi-core-friendly and memory-bandwidth-efficient processing scheme. On the other hand, recently, graphics processing units (GPUs) have emerged as powerful tools for general high-performance computing. GPUs are affordable and energy-efficient devices that deliver a massive computational power by utilizing a large number of cores and a high memory bandwidth. GPUs can be used as co-processors for query acceleration of in-memory databases. One of the main bottlenecks in GPU-acceleration of in-memory databases is the need for data to be transferred back and forward between GPU memory and RAM through a low-bandwidth PCIe bus. To address this problem, in this study, a new generation of in-memory databases is proposed that instead of keeping data in main memory stores it in GPU device memory.

1. INTRODUCTION

The amount of digital information in our world is growing in a very fast pace. Enterprise companies generate thousands of terabytes of structured data every day. This data needs to be analyzed as fast as possible in order to maintain the competiveness of the company. In many scientific projects, generating multi-terabyte data sets in a daily bases has become a very common task.

In order to explore the interesting patterns in these massive structured datasets, scientists need to be able to find the answer of their ad hoc queries interactively. The traditional models and tools of computation can no longer handle this massive demand for data processing. Investigating new methods and developing new tools for fulfilling this need has become one of the mainstream areas of research in recent years.

The ongoing research in the area of processing large data collections falls into two major categories. First category consists of a variety of MapReduce framework [5]. This framework is designed to distribute the data collection over a large cluster of commodity machines and process it in a fault-tolerant fashion. This computational model is suitable for non-interactive purposes in which response time is not an issue. Due to the simplicity and flexibility of this model for processing both structured and unstructured data, it has attracted too much attention in recent years. Although this model is pretty scalable, because of its significant computational overhead, it underutilizes the resources available in the system significantly. In addition, maintaining large clusters of commodity machines is costly and power-inefficient. Small companies who cannot afford to maintain a large cluster of machines can use the cloud computing services provided by third party companies such as Amazon. However, in many cases transferring large data sets generated in a daily bases to a third party cluster is not feasible. In addition, due to the security considerations some companies are not willing to upload their data on servers of a third party company.

Second category consists of in-memory column-oriented databases. The focus of in-memory databases are mainly on processing structured data. Although this model is not as scalable as MapReduce model, it benefits from SQL language which is a very well-known and well-established query language. Compared to MapReduce model, main-memory databases are much more resource-efficient. Therefore to reach the same computational power, we need a smaller number of resources and a lower amount of power. Due to technological advancements, the cost of RAM is decreasing and the amount of memory that can be used in a single machine is increasing. These trends tend to continue in the future. Currently, one can buy a server equipped with a quarter of terabyte of RAM for less than $20K.

Due to considerable increase in demand for fast analysis of structured data, in-memory databases have attracted the attention of large database vendors in recent years. TimesTen [4] (acquired by Oracle in in 2005), SolidDB [5] (acquired by IBM in 2008), VoltDB [6] (announced in May 2010) and HANA [3] (announced on June 2011 by SAP) are a few examples to name. Monet-DB [7] is an open-source and mature in-memory database that addresses array-based scientific data processing as well.

The main advantage of in-memory databases over traditional databases is that they eliminate the disk I/O barrier by storing the data in memory. Most in-memory databases employ a column-oriented data layout. Column-oriented data layout is cache-efficient, multi-core friendly and compression-friendly. Therefore, in these databases, for most analytical queries, instead of the I/O, the CPU gets a bottleneck.

After that the traditional CPU evolution did hit the power wall in about 2001, increasing the clock rate of CPUs became almost impossible. Since then, the multi-core architecture has become the dominant solution in processing. In-memory databases can generally utilize the multi-core parallelism much better than traditional databases.

Since about 2007, graphics processing units (GPUs) have emerged as powerful tools for high-performance general-purpose computing. Compared to a cluster of commodity machines, GPUs deliver an equivalent performance at one tenth of the cost and one twentieth of the power consumption. GPUs provide efficient inter-processor communication through on-chip local shared memory; present an interesting amount of memory bandwidth; support the concurrent execution of thousands of threads with a very low context-switch overhead; and support general purpose parallel programming models which make the process of developing parallel programs very convenient. New capabilities of the GPUs make them the economic and high-performance many-core processing machines that can be used to accelerate a wide range of computationally intensive applications.

Because of these outstanding capabilities, GPUs have enthusiastically received attention recently in the area of scientific research. Many researchers are studying the application of the new capabilities of the GPUs in various computationally intensive scientific areas like physics simulation, molecular dynamics, bioinformatics, computational finance, image processing, medical engineering, pattern matching, video encoding, etc.

Investigating the possible applications of GPUs in data processing is another new and hot topic of study in recent years. So far, in this direction, only a small number of strong scientific works have been done and there exists still a great opportunity for further research. Most of the studies have considered the previous generation of GPUs and have not employed the new features provided in the latest generation. In addition, in most studies, only a single GPU machine has been targeted while multi-GPU machines and also clusters of GPU-enabled machines are emerging rapidly.

Utilizing the computational power of GPUs in data processing seems to be a promising idea. However, GPU computing suffers from a number of limitations. In GPU computation model, the data needs to be transferred back and forward between GPU memory and RAM through a low bandwidth PCIe bus. The bandwidth of PCIe 2.0 is about 25 times less than the bandwidth of GPU memory. In applications in which data transfer between GPU and CPU is inevitable, this bottleneck results in a considerable underutilization of the computational power of GPU. In some cases, the cost of transferring data to GPU memory is more than the benefit of GPU acceleration. Every GPU consists of a number of streaming multi-processor units (SMs). Each SM is basically a SIMD unit which is supposed to execute a single instruction on a large contagious part of memory. Therefore, memory fragmentation and branch divergence are two important issues that have a considerable negative impact on utilization of the computational resources in GPU. The way in which a program assigns threads to SMs of the GPU is another factor that plays an important role in GPU utilization. In addition, GPUs employ a multi-level memory hierarchy which is manually controllable by the programmer. All these architectural details not only add too much complexity to GPU programming, but also make the use of GPU unsuitable for many applications. For instance, suppose that we would like to implement the "scan" operator of a traditional row-oriented database on GPU. Since the data layout is row-based, in order to evaluate a predicate on a single column, we need to transfer the entire table into GPU memory. As noted before, the cost of this transfer can simply outperform the benefit of GPU acceleration. Therefore, there might not be any point in using a GPU to accelerate the scan operator in a traditional database. If we employ a column-oriented data layout, the cost of data transfer will be reduced significantly. It is because in this case we need to transfer only a single column to GPU memory. We can utilize very effective compression technique to reduce this cost even more. Thus, it should be considered that the GPU acceleration is not applicable in any arbitrary computing problem.

Even in applications that are good candidates for GPU acceleration, for writing an optimized code, a programmer needs to be familiar with the architectural details of the GPU. Compared to CPU, performance in GPU is much more sensitive to programming mistakes. It is known that a simple programming mistake in GPU programming might result in about two order of magnitude performance degradation.

To address the programming complexity of GPUs, one idea is to develop some higher level general programming frameworks for users. In this way the user does not need to be familiar with all architectural details of the GPU. However, it is a rule of thumb that the simplicity and generality always come at the cost of performance. A general programming framework lowers the degree of user control over the system resources. Therefore, the chance of underutilization of the computational power of GPU increases. In [11, 12, 21, 25, 28, 31] some variations of GPU-enabled MapReduce framework are proposed. From these works it can be observed that the resulting programming frameworks do not perform efficiently for solving any arbitrary data processing problem. Although the MapReduce is a scalable and fault-tolerant framework, its scalability and fault-tolerance comes at the cost of a huge amount of computational overhead. This computational overhead results in the underutilization of the computational power of cluster. Due to aforementioned hardware restrictions in GPU, if we employ the MapReduce computational model on GPU, the amount of underutilization of the system becomes even more considerable. Therefore, applying the MapReduce paradigm into GPU computing for general processing of large data collections is not an efficient approach.

Leveraging in-memory databases is an alternative method for efficient processing of large data collections. This method has been around for a while, but because of improving the cost-effectiveness of main memory in recent years this approach has become more promising. Due to high level of parallelizability in in-memory databases and because of the high memory bandwidth of GPUs, in-memory databases can benefit a lot from GPU acceleration. In this way the user can take advantage of the computational power of GPUs using SQL, a high level, expressive, and well-known query language, without dealing with the architectural details and the programming complexity of the GPU computing.

The main idea of this study is to introduce a new generation of in-memory databases for processing analytical workloads that utilizes the "GPU Memory" instead of RAM for data management, to eliminate the data transfer time between GPU memory and RAM. The ever-increasing amount of GPU memory in the new generation of GPUs, the possibility of using multiple GPUs in a single machine, and the emergence of GPU-enabled clusters, all serve to make this idea even more promising. Currently, the maximum memory capacity of a single board GPU is 8 GB in Tesla K10 from Nvidia. In Q4 of 2012, Nvidia will announce Tesla K20 which is supposed to have 12 GB of memory. The memory capacity tends to continue its increasing trend in future generations. Today, one can easily put four of these GPUs in a single commodity server. Moreover, the memory capacity can be increased by employing a cluster of these commodity servers. Thus, the idea of storing the entire data in GPU memory is becoming more feasible day by day.

The rest of this paper is organized as follows. In section 2 previous studies on utilizing the computational power of GPUs in data processing is reviewed. In section 3 the architectural details and the programming model of GPU is explained. Finally, in section 4 the details of the proposed research plan are discussed.

2. Literature Review

MapReduce [10] is a distributed programming model, introduced by Google, for large-scale parallel computations. In this model, each Map function takes a portion of data as input to generate a list of intermediate key-value pairs. Each Reduce function receives a key and the list of all values associated with that key and generates a single value or a list of results. In MapReduce, the developer needs only to develop the Map and Reduce functions; all other complex tasks, such as scheduling, synchronization, load-balancing and fault tolerance, are handled automatically by the framework. The simplicity of the programming model, along with the quality of services provided by the MapReduce framework, has created considerable enthusiasm and many success stories in the large-scale processing community.

Thus far, only a few implementations of the MapReduce framework for GPUs have been proposed. Mars [21] is a MapReduce framework was designed in 2008 to run on a single GPU. Although the Mars framework is not scalable to more than one GPU, it is the first project that attempts to implement a MapReduce programming model for GPUs to simplify parallel programming. Mars is up to sixteen times faster than Phoenix, the best known multi-core CPU-based implementation of MapReduce, and up to 7 times faster than Mars-CPU, the multi-core CPU based implementation of MapReduce proposed by the authors, in a number of common web analytical applications.

Catanzaro et al., in 2008 proposed another implementation of the MapReduce framework for a single GPU [8]. They made some changes in the structure of the Map function and introduced a new concept called "predicates". Their framework is up to 34 times faster in training the Support Vector Machine and up to 150 times faster in classification task than the best CPU version. DisMaRC [25], the distributed version of Mars, runs on a small cluster of two CPU-GPU nodes with two Nvidia FX5600 GPUs attached to each node. DisMaRc showed that a speedup of more than four times that of Mars is achievable. The DisMaRC did not conduct performance tests against a standard cluster running a conventional MapReduce, but it showed that the initial Mars research was on the right track.

In 2009, Farivar et al. [11] introduced a special-purpose variation of the MapReduce framework based on Hadoop [2] (an open-source implementation of Google MapReduce) that utilizes the computational power of GPUs in the Map and Reduce functions. This work demonstrated, for the first time, that a small five-node CPU-GPU cluster could outperform a regular 62-node CPU cluster and achieve a four times faster execution when performing the Black Scholes option pricing algorithm. Although this work shows the significant potential of CPU-GPU hybrid clusters, the Black Scholes option pricing problem is a specific problem in which the level of independency of parallel sub-tasks is substantial. In addition, in this work, the MapReduce programming model is tailored to the specific characteristics of a particular problem. Thus, the proposed framework is not simply applicable to other common analytical tasks. The results of this paper, therefore, do not necessarily hold for other common analytical problems.

The authors of the Mars framework, in 2011, proposed MarsCUDA[12], an advanced version of their previous work, Mars. MarsCUDA extends the capabilities of Mars and employs a number of optimization techniques to improve the performance. MarsCUDA can be used on a multi-GPU system. A variation of MarsCUDA is able to utilize the computational power of the CPU and GPU at the same time. They have also implemented a variation of MarsCUDA, called MarsHadoop which is basically a GPU-enabled version of the Hadoop Framework. A multi-core CPU version of the MapReduce framework called MarsCPU is also proposed and it is shown that it outperforms Phoenix [28], the best known multi-core CPU implementation of the MapReduce framework, in all considered benchmarks. MarsCUDA shows up to 25 times speedup over Phoenix and up to 10 times speedup over MarsCPU. MarsHadoop, versus Hadoop, shows a 2.5 times speedup on a cluster of two GPU-enabled nodes. In this work, the authors have also implemented MarsBrook on AMD GPUs using the stream programming model Brook+[1]. Due to the limitations of Brook+, however, MarsBrook is less advanced than MarsCUDA in terms of the performance and expressivity.

Stuart and Owens in [29] proposed another Multi-GPU MapReduce framework called GPMP for GPU Clusters. In Mars, there is a one-to-one relationship constraint between data items and GPU threads while in GPMP, by relaxing this constraint, a much more flexible MapReduce framework has been proposed. The many-to-one or one-to-many assignment of data items to threads, allows developers to exploit the inter-thread cooperation inside a mapper to increase the performance. For example, in GPMP, a parallel prefix scan can be done inside a mapper. Although this approach increases the flexibility of the framework significantly, this extra flexibility comes at the cost of extra complexity that is in contrast to the main goal of the MapReduce framework. In order to reduce the required communication over network and PCIe bus, a number of additional stages such as partial reduction, accumulation and combine can be employed in MapReduce pipeline of the GPMR. The developer, based on the characteristics of the task, can employ a combination of these additional stages. In additions, sorting and partitioning phases of the MapReduce pipeline can be optimized by developer for particular workloads. The efficiency of GPMR using five different benchmarks has been analysed. The experiments demonstrate a speedup of up to 162 times over phoenix and up to 37 times over the first version of the Mars on a single GPU. Unfortunately GPMR has been not compared to a regular CPU-based version of the MapReduce such as Hadoop. Since this work has been published at the same time as the MarsCUDA, no comparison between MarsCUDA and GPMR has been done so far.

In all aforementioned variations of the MapReduce framework which are developed to simplify the programming for GPUs, it can be observed easily that the amount of the expected performance improvement resulted from GPU acceleration highly depends on the characteristics of the task. The degree of the effectiveness of GPU depends on many parameters, including the inter-dependency of sub-problems, the amount of data transfer required in the solution, the size of the problem compared to the available global memory in the GPU, the locality of memory, the computational intensity of the independent sub-problems, and finally the design decisions of the MapReduce implementation.

In [9, 16, 17, 22, 24, 29, 30] the computational power of GPU is exploited to accelerate sorting as one of the fundamental operator in any data processing algorithm. Although all these works show interesting speed-ups compared to CPU, since the degree of data dependency in sorting algorithms is pretty high, the amount of speed-up is not very significant. In [23] it is claimed that after applying optimizations appropriate for both CPUs and GPUs, CPU will even outperform the GPUs in sorting throughput. In this paper the high speed-ups reported in other papers are attributed to the fact that in those papers an optimized GPU code is compared to an unoptimized CPU code or a high-end GPU is compared to a commodity CPU.

In [14, 15, 19, 20] GPU is used as a co-processor for accelerating query processing in a traditional database. A variety of primitive operators such as map, scatter, gather, reduce, prefix scan, split, filter and sort are addressed for GPU acceleration and based on these primitive operators, higher level operators such as select, join, aggregate, group by, and order by are defined in [19, 20]. On in-memory data the GPU-based primitives and high level query processing algorithms achieve a speedup of 2–27X over their optimized CPU-based counterparts. However, considering the data transfer time between the CPU and the GPU, the GPU-based algorithms achieve a 2–7X performance speedup over their CPU-based counterparts for complex queries such as joins, and 2–4X slow down for simple queries such as selection. It is shown that the data transfer between CPU and GPU can contribute to 15–90% of the total time in relational query co-processing.

An OLTP engine for high throughput transaction processing in in-memory databases called GPUTx is proposed in [18]. In this approach multiple transactions are grouped in a batch and are executed concurrently on GPU as a single task. They shows that by utilizing this bulk execution model, compared to a quad-core CPU, the optimized GPUTx achieves 4-10 times higher throughput in transaction processing.

In order to reduce the data transfer between GPU and CPU in GPU query co-processing, in [13] a number of lightweight compression schemes are proposed. It is shown that by combining different compression schemes higher compression rates are achievable. They introduce a compression optimizer that chooses the best combination of compression schemes smartly. By utilizing the resulting compression the data transfer between CPU and GPU is reduced significantly and the overall performance of the query co-processing will be improved. There are some similarities between this work and our proposed approach. For instance they have integrated their GPU-based compression into MonetDB. However, this work is based on this assumption that the data will be stored in main memory and the GPU will be used as a co-processor for accelerating the query processing in an in-memory database, while our proposed approach is based on this assumption that the data would be stored in GPU memory and therefore there is no need to load the date into GPU memory before processing. Nonetheless, the compression schemes proposed in this paper can be utilized in our approach in order to reduce the size of the data stored in GPU memory.

Using field programmable gate arrays (FPGAs) [26, 27, 32] is another solution for accelerating data processing. Success of companies such as Netezza [27] in this area confirms the applicability of this approach. However, designing special-purpose components using FPGAs typically needs a team of embedded systems programmers, hardware designers, and system software developers with a considerable expertise with system level architecture. This makes the cost of building and maintaining such systems quite high. On the other hand, thanks to mass production in gaming industry, GPUs are available in every machine with a very affordable price. In addition, their performance, feature set, memory capacity, and power efficiency is improving generation by generation. Therefore, unlike FPGA-based solutions, in GPU-based solutions there is no need to be worry about the hardware. Moreover, powerful and flexible programming frameworks such as CUDA and OpenCL are designed to simplify GPU programming for every programmer.

3. GPU architecture and Computing Model

GPUs are originally designed for accelerating graphics processing by employing fixed hardware pipelines for rendering. Due to the massive computational power, cost effectiveness, energy efficiency and rapidly improving programmability, they have recently become a powerful tool for general purpose computing.

Modern GPUs consist of a number of SIMD units with a wide SIMD length. These SIMD units are usually called streaming multi-processors (SMs). Inside each SM there are several scalar processors (cores). Cores in GPUs are simple processing units which are capable of performing basic arithmetic and logic instructions. The main difference between GPUs and multicore CPUs is that in GPU there are a large number of simple cores while in CPU there are a small number of complex cores. That is why GPUs are called many-core processors. For instance, in Nvidia Tesla K10 there are 16 streaming multi-processors with 192 cores in each. In other words, unlike CPUs, in GPUs a larger number of transistors are dedicated to processing rather than control. Since CPUs are designed for processing sequential programs, in CPUs a larger portion of the processor is devoted to complex control mechanisms such as out of order execution and branch prediction. In GPU programming, this is the programmer who is in charge of optimizing the control flow of the program. Therefore, typically, writing an optimized code for GPU is much more difficult that writing an optimized code for CPU.

GPUs are equipped with a high bandwidth and high access latency device memory. The memory bandwidth of the GPU is about 8-10 times the memory bandwidth of CPU and at the same time its access latency is about an order of magnitude higher than that of CPU. The high access latency of GPU will be compensated by loading a massive number of threads in GPU cores. In this way, whenever a thread is waiting for a memory access instruction, other threads can be scheduled for execution. Thus, in order to keep all cores busy, a GPU program needs to employ a large number of threads.

Every SM in GPU employs a small amount of low latency memory which is called shared memory. Shared memory is accessible by all threads within a SM and is used for fast inter-communication between threads which are located at the same SM. It also can be used as a managed cache to reduce the number of expensive memory accesses to device memory whenever it is possible. In GPU, multiple load/store instructions issued by consecutive threads to consecutive memory locations would be combined to a single load/store instruction. This optimization technique is called memory coalescing. Memory coalescing has a considerable positive impact on the performance and utilization of the program. In column-store databases, values of the same columns are located in physically consecutive addresses in memory. Therefore, the memory layout of the column-store databases is perfectly aligned with the computational model of the GPU.

A typical GPU program consists of three stages. In first stage, CPU uploads data from RAM to GPU device memory. In second stage, CPU will ask GPU to process the uploaded data and to store the results in GPU device memory. In third stage, the CPU will download the results of the computation from GPU device memory into RAM. As mentioned before, in stages one and three, data needs to be transferred between RAM and GPU device memory through a low-bandwidth PCIe bus. The low bandwidth of PCIe bus is an important bottleneck which can degrade the overall performance of the GPU programs significantly. By storing the entire data in GPU memory this data transfer time can be eliminated.

4. Research Plan

The main goal of this research is to show the applicability of an in-GPU-memory column-oriented database. To the best of our knowledge, using GPU memory instead of RAM in an in-memory database is a novel approach which is not studied yet. In this research we plan to modify an existing open-source in-memory database called MonetDB [7] in such a way that it uses GPU memory instead of RAM for data management. The modifications will be done in three phases. The goal of the first phase is to utilize the GPU memory of a single GPU machine; in the second phase, the database engine would be able to utilize multiple GPUs on a single machine; and in the third phase a distributed in-GPU-memory database will be developed which is able to utilize the GPU-memories and the computational power of all GPUs in a cluster of GPU-enabled machines. In this way, after providing a proof of concept in phases one and two, the scalability of the approach will be addressed in phase three.

The target of this system is analytical data processing workloads. We are interested in answering complex filtering and aggregations over large datasets. Minimizing the query response time by improving the utilization of the computational power of GPUs is the most important goal in developing the system.

One explicit assumption in developing this system is that the data collection is read-only and it would be loaded into GPU memory before issuing the queries. Although some in-memory-updates might be allowed to facilitate the multi-stage data processing, those updates are not supposed to be reflected on a stable storage.

There are a number of research problems that would be addressed during the course of this study. In particular, we are interested in working on the following problems in our setting:

- Query optimization for GPU architecture: considering the memory hierarchy, resource limitations and other architectural constraints of the GPU in query optimization
- Efficient load balancing over multiple GPUs: how can we balance the load efficiently over a multi-GPU setting considering the bandwidth of the links between different GPUs located in the same machine as well as the bandwidth of the different GPUs located in different machines
- Efficient coordination, synchronization, and communication among GPUs: How to coordinate the data flow among multiple GPUs, how to minimize the synchronization time by employing lock-free synchronization mechanisms, and how to minimize the required communications between GPUs efficiently
- Effective data compression: how to employ data compression technique to save storage capacity and to reduce the communication cost among multiple GPUs during the query execution
- Multi-stage query processing: study the feasibility of storing the intermediate results in GPU memory in multi-stage query processing considering the memory capacity constraints
- Concurrent query processing: how to improve the throughput of the system by answering a batch of multiple queries
- Hybrid GPU-CPU computation: how to exploit the computational power of CPU and GPU at the same time to accelerate the query processing in a hybrid setting

To evaluate the performance of the system, we will use query response time and GPU utilization metrics. We will consider a number of realistic large data sets and their corresponding query collections and compare the performance of the system over those collections before and after applying the proposed ideas. We will also perform a number of experiments to investigate the behavior of the system and to evaluate its advantages and disadvantages quantitatively. For instance we will perform an experiment over different number of nodes to study the scalability of our approach after developing the third phase.

5. REFERENCES

[1] AMD Brook+ http://developer.amd.com/documentation/articles/pages/ getextraordinaryperformancebyexploitingthegpu.aspx, 2012

[2] http://hadoop.apache.org, 2012

[3] http://www.sap.com/solutions/technology/in-memory-computing-platform/hana/overview/index.epx, 2012

[4] http://www.oracle.com/us/products/database/timesten/overview/index.html, 2012

[5] http://www-01.ibm.com/software/data/soliddb, 2012

[6] http://voltdb.com/, 2012

[7] http://www.monetdb.org/Home, 2012

[8] Bryan Catanzaro, Narayanan Sundram, Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors, workshop on Software Tools for MultiCore Systems, 2008.

[9] Daniel Cederman and Philippas Tsigas. 2010. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. J. Exp. Algorithmics 14, Article 4 (January 2010)

[10] Jeffrey Dean and Sanjay Ghemaw. MapReduce: simplified data processing on large clusters, 6th conference on Symposium on Opearting Systems Design & Implementation, 2004.

[11] Reza Farivar, Abhishek Verma, Ellick Chan, Roy H. Campbell. MITHRA: Multiple data independent tasks on a heterogeneous resource architecture., 2009 IEEE International Conference on Cluster Computing, 2009.

[12] Wenbin Fang and Bingsheng He and Qiong Luo and Naga K. Govindaraju. Mars: Accelerating MapReduce with Graphics Processors, IEEE Transactions on Parallel and Distributed Systems, 2011.

[13] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. Proc. VLDB Endow. 3, 1-2 (September 2010), 670-680.

[14] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2007. GPUQP: query co-processing using graphics processors. In Proceedings of SIGMOD '07, ACM, New York, NY, USA, 1061-1063.

[15] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast computation of database operations using graphics processors. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04). ACM, New York, NY, USA, 215-226.

[16] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In Proceedings of SIGMOD '06, ACM, New York, NY, USA, 325-336.

[17] Linh K. Ha, Jens Kruger, Claudio T. Silva. Fast Four-Way Parallel Radix Sorting on GPUs Comput. Graph. Forum, 28(8), 2368-2378, 2009

[18] Bingsheng He and Jeffrey Xu Yu. 2011. High-throughput transaction executions on graphics processors. Proc. VLDB Endow. 4, 5 (February 2011), 314-325.

[19] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In Proceedings of SIGMOD '08, ACM, New York, NY, USA, 511-524.

[20] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. ACM Trans. Database Syst.34, 4, Article 21 (December 2009), 39 pages.

[21] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors, 17th international conference on Parallel architectures and compilation techniques (PACT '08), 2008.

[22] Bonan Huang, Jinlan Gao, Xiaoming Li. 2009. An Empirically Optimized Radix Sort for GPU. In Proceedings of the International Symposium on Parallel & Distributed Processing with applications

[23] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News 38, 3 (June 2010)

[24] Duane G. Merrill and Andrew S. Grimshaw. 2010. Revisiting sorting for GPGPU stream architectures. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10). ACM, New York, NY, USA, 545-546.

[25] Alok Mooley, Karthik Murthy, Harshdeep Singh. DisMaRC: A Distributed Map Reduce framework on CUDA, tehcnical report, University of Texas at Austin, 2009.

[26] Rene Mueller, Jens Teubner, and Gustavo Alonso, Data processing on FPGAs. Proc. VLDB Endow. 2, 1 (August 2009), 910-921.

[27] Netezza, http://www.netezza.com/

[28] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems, 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, 2007.

[29] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In Proceedings of the 2010 international conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 351-362.

[30] Nadathur Satish, Mark Harris, and Michael Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In Proceedings of IPDPS '09, Rome, Italy, May 23-29

[31] Jeffery Stuart, John D. Owens, Multi-GPU MapReduce on GPU Clusters, IPDPS, 2011

[32] Louis Woods, Gustavo Alons, Fast data analytics with FPGAs, ICDE Workshops 2011: 296-299.