GPU Accelerated Array Queries: The Good, the Bad, and the Promising

Feng Liu, Kyungyong Lee, Indrajit Roy, Vanish Talwar, Shimin Chen, Jichuan Chang, Parthasarthy Ranganathan

HP Laboratories
HPL-2014-50

Keyword(s):

GPU; SciDB; Database

Abstract:

Array databases are well suited for complex multidimensional analysis. However, array queries are often performance constrained by the high computational demands of the underlying algorithms. We explore the use of GPU to accelerate these algorithms and study its end-to-end effects on performance, power, and energy efficiency. We have extended SciDB, a popular array database, to use GPUs and improved its query performance by 1.5X to 11X. While GPUs improve both performance and energy efficiency, multiple design issues limit us from reaching the touted 100X performance benefits of GPUs. We provide detailed experimental analysis to understand these bottlenecks related to array partitioning, load imbalance, and CPU-GPU hybrid execution.

External Posting Date: October 6, 2014 [Fulltext] Approved for External Publication
Internal Posting Date: October 6, 2014 [Fulltext]

GPU Accelerated Array Queries: The Good, the Bad, and the Promising

Feng Liu Kyungyong Lee Indrajit Roy Vanish Talwar
Shimin Chen Jichuan Chang Parthasarthy Ranganathan
HP Labs

Abstract

Array databases are well suited for complex multi-dimensional analysis. However, array queries are often performance constrained by the high computational demands of the underlying algorithms. We explore the use of GPU to accelerate these algorithms and study its end-to-end effects on performance, power, and energy efficiency.

We have extended SciDB, a popular array database, to use GPUs and improved its query performance by 1.5× to 11×. While GPUs improve both performance and energy efficiency, multiple design issues limit us from reaching the touted 100× performance benefits of GPUs. We provide detailed experimental analysis to understand these bottlenecks related to array partitioning, load imbalance, and CPU-GPU hybrid execution.

1 Introduction

Array oriented databases (AODB) are increasingly being used to process large amounts of multi-dimensional data. AODBs store logical arrays and provide SQL-based query languages with mathematical operators (SciDB [4], RasDaMan [3], MonetDB [1]). Typical queries in array databases involve dense and sparse linear algebra (for machine learning and graph analysis), and moving window operations (for image analysis). These operations are computationally demanding, involving $O(n^2)$ or even $O(n^3)$ operations. Given the complexity of these algorithms and the increasing sizes of scientific and sensor data, array queries can become very expensive to run on commodity processors.

A promising approach to improve query performance is to offload expensive computation from the CPU to hardware accelerators such as NVIDIA GPU, AMD APU, or Intel MIC [11]. Many array computations easily map to vector operations and, hence, can naturally leverage the massive parallelism provided by accelerators. To validate our hypothesis that accelerators can be beneficial, we measure the best case speedups in four array applications when executed on an NVIDIA Tesla GPU instead of a single Xeon core. Figure 1 depicts the speedups seen on the GPU. Application and dataset details are in Table 1. Since we are interested in the best case performance improvement, we only measure the GPU kernel time of the application. All other overheads, such as the time to load data from disk, decompress data, transfer data between the host and the GPU, and so on, are excluded. The plot shows the potential of GPU acceleration: the crux of the computations can run 26× to an astounding 1,400× faster!

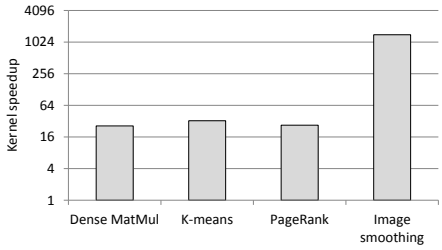


Figure 1: Application kernel speedups: GPU vs. single-core CPU implementation. Y-axis is log scale.

To explore these benefits, we have modified SciDB, an array database, to leverage GPUs. Our results reveal many interesting and non-obvious observations that can guide future research in this area. For example, in certain applications a GPU can provide 100× end-to-end performance gains over single-core CPU implementations. However, the gains become 1.5× to 10× against parallel CPU implementations utilizing 16 CPU cores. In addition, data representation and partitioning schemes significantly impact performance. Small array chunks are better for CPU parallelism while large array chunks reduce the overhead of loading data and transferring to the GPU. Furthermore, unlike popular belief, moving data between host and GPU is not always the biggest overhead. After offloading computation to the GPU, time spent inside

Application	Input data	Application Characteristics		
		Complexity	Parallelism	Ops/element
Dense MatMul	Two n×n matrices: n=25,600, 2×10 GB	$O(n^3)$	Parallel across chunks	n
K-means clustering	n 3-D points: n=100M, K=512, 2.3 GB	O(K·n)	Reduction at end	K
PageRank	Graph: v=33M, e=286M, 4.5 GB	O(e)	Parallel across chunks	v
Image smoothing	n×n image: n=51,200, w=7×7, 20 GB	$O(w \cdot n^2)$	Parallel across chunks	w

Table 1: Application characteristics and their input data.

the database engine or in converting the data format may overshadow the GPU kernel execution time, and in some cases, even the data transfer time. Thus, depending upon the execution mode (CPU, GPU, or both), different design knobs need to be dynamically tuned in coordination.

2 Background

SciDB is an open source array management and analytics database [4]. It has a shared-nothing, distributed architecture for array storage and analysis. SciDB has a coordinator node and multiple worker nodes. Array data is partitioned into and stored as equal sized sub-arrays called *chunks*. Chunks are the basic unit for I/O, computation, inter-node communication, and version control. Chunk sizes are specified by users and can be tuned to improve parallelism and reduce memory access latency [12]. To reduce storage overhead chunks are compressed using a variant of run length encoding (RLE). SciDB supports both sparse and dense arrays, and can store array chunks that overlap. For example, graphs are stored as sparse matrices while images are stored as dense chunks with overlapping boundary elements.

We evaluate four representative applications- matrix multiplication, PageRank, K-means, and image smoothing (Table 1). Our choice of applications is motivated by the four classes of operations that array databases support: dense array operations, sparse array operations, user-defined functions, and window based aggregation. Not only do these applications exercise different SciDB components, they also have different computational complexity and parallelism characteristics.

3 System design

Figure 2 shows our overall framework integrating GPUs in SciDB. We describe specific components below.

GPU scheduling and monitoring. We have added a scheduler to decide when the GPU should execute computation tasks. After putting tasks in a work queue, executor threads remove tasks from the queue and process them on CPU or GPU based on the scheduler policy. The scheduler can use static policies, such as a fixed limit on the maximum number of concurrent GPU tasks, or use runtime metrics, such as GPU utilization, to tag tasks.

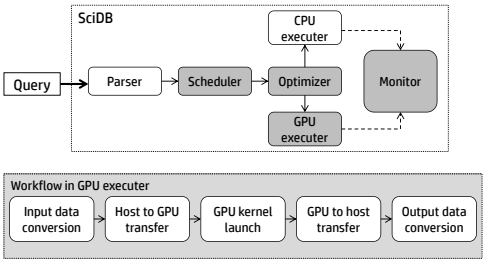


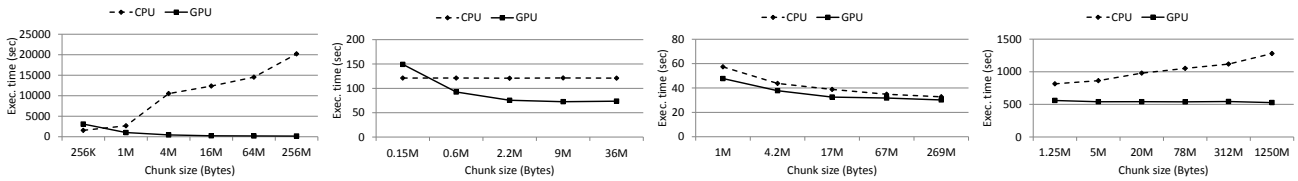
Figure 2: GPU offloading in SciDB.

Dynamic chunk optimization. An array chunk or sub-array is the basic storage and computation unit in SciDB. Chunk sizes thus have a significant impact on data transfer time (from disk, or between host and GPU) and execution time on the CPU or the GPU. We modified SciDB's optimizer to support chunk resizing. Once the scheduler determines where chunks will be processed (CPU or the GPU), the optimizer divides or merges chunks. Our prototype does not automatically find optimal chunk sizes. Instead, we configure the optimizer to use a target chunk size at runtime, and study the effect of resizing chunks on application performance.

GPU execution. We have modified SciDB's execution engine to incorporate GPU kernel launches. We added support for three kinds of data conversions. First is *data structure conversion* to convert data from an array-of-structures in SciDB to the GPU preferred structure-of-arrays. For example, in PageRank, sparse array representation of the graph has to be converted from SciDB's list of values to compressed sparse row format. It helps coalesce GPU memory accesses. The second type of data conversion is *matrix orientation conversion* which changes the orientation of multi-dimensional matrix. For K-means, we use this type of conversion to transpose the row major ordered input on the CPU side to column major ordered GPU kernel input. The final conversion type is *data decompression* where compressed data stored in SciDB is converted to flat uncoded data.

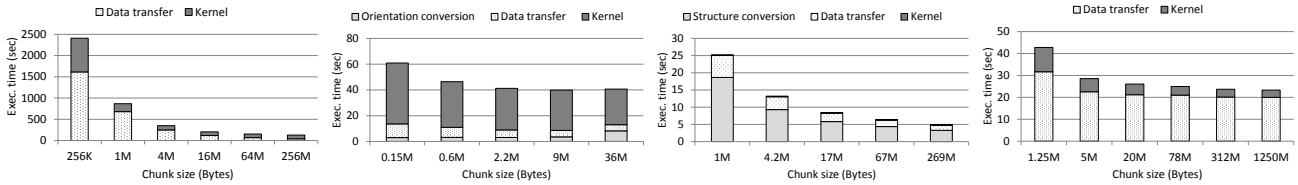
4 Evaluation

All experiments are run on an HP SL250 server with Ubuntu 11.10. The server has two Intel Xeon E5-2650 2.0 GHz processors (total of 16 cores), 128 GB DRAM,



(a) Dense matrix multiplication (b) K-means clustering (c) PageRank (d) Image smoothing

Figure 3: Effect of chunk size on total execution time when using 16-core CPUs or a GPU. Lower is better.



(a) Dense matrix multiplication (b) K-means clustering (c) PageRank (d) Image smoothing

Figure 4: Overheads when using a GPU. Lower is better. The execution time only includes the GPU kernels and related overheads, such as data movement and data conversion, to enable the kernels.

and a 200 GB SSD. We attached a NVIDIA Tesla M2090 GPU (total of 512 cores). The GPU has 6 GB global memory and peak CPU-GPU bandwidth is 8 GB/s on PCIe 2.0. For dense matrix multiply, PageRank, and image smoothing, we use cublas, cusparse libraries in CUDA 5.5, and image convolution library. We wrote our own GPU implementations for K-means on SciDB 12.3.

4.1 Effect of array chunk sizes

CPU case. Figure 3 evaluates the effect of different chunk sizes on the CPU and GPU implementations of the applications. When using the CPU configuration, the execution time of all applications, except PageRank, increases with increase in chunk size. For example, in dense matrix multiplication, chunk size of 256K gives 84% better performance compared to chunk size of 256M. The performance difference is due to spatial locality. In matrix multiplication, each chunk is stored in row order but during multiplication the chunks of the right hand side matrix is accessed in column order. Uniquely, PageRank shows less spatial locality effects. PageRank accesses a relatively small vector in random order while the larger adjacency matrix is accessed sequentially.

GPU case. Unlike the CPU case, performance of GPU implementations of applications improve as chunk size increases. For example, matrix multiplication is 19× faster for chunk sizes of 256M compared to chunk sizes of 256K (Figure 3a). Similarly, PageRank's performance is 37% better at 269M chunk size compared to 1M chunk size. With large chunks, the transfer overhead between host and the GPU decreases substantially. Few large data transfers are invoked instead of multiple small transfers. Figure 4 reveals that both data transfer and computation time decreases with increase in chunk size. Unlike in the CPU case, single core data locality has a minor effect in the GPU device because different cores anyway have to fetch data from the device memory.

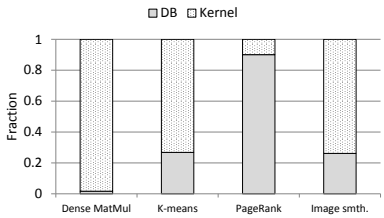
Observation 1: Smaller chunks preserve spatial locality and improve performance of CPU implementations.

Observation 2: Larger chunks increase GPU application's performance due to lower scheduling and data transfer overheads.

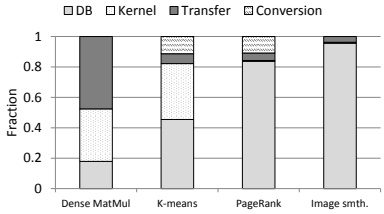
4.2 Overheads in using a GPU

Offloading computation to the GPU changes which software components become performance bottlenecks. When using 16 CPU cores (Figure 5a), the main application kernel takes up more than 73% of the total time in matrix multiplication, K-means, and image smoothing. Figure 5b shows that after GPU offloading, less than 37% of the time (for some applications less than 1%) is spent in the main kernel. In many cases the database itself, such as SciDB code to read data and iterate through data-structures, become a major bottleneck.

Data transfer. Data transfer between host and GPU is an overhead present in all applications. Figure 5b shows the breakdown of execution time spent in different components. Of the total execution time, data transfer takes upto 47% in dense matrix multiply, 6% in K-means, 5% in PageRank, and 4% in image smoothing. It is startling to see that in PageRank and image smoothing, the actual compute kernel takes less than 1% of the total execution



(a) CPU



(b) GPU

Figure 5: 16-core CPU vs. GPU processing: Fraction of time spent in different components.

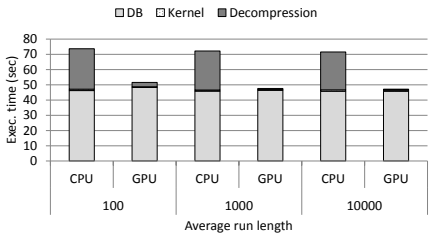


Figure 6: RLE decoding in 16-core CPU vs. GPU.

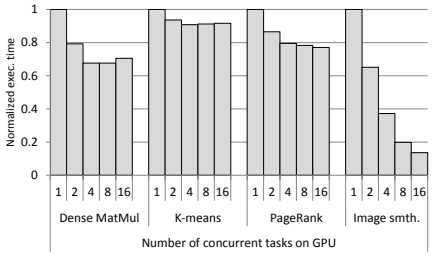


Figure 7: Performance of GPU in shared mode.

time. In contrast, dense matrix multiply and K-means are compute bound and spend 34% and 37% respectively of their total execution time on actual computation.

Data conversion. Figure 5b shows that 11% of the time in K-means and 10% in PageRank is spent in data conversion in the CPU. In K-means the input matrix is converted to column major order while in PageRank the input matrix is transformed to compressed sparse row format.

Decompression. Decompression overhead depends upon input data: our randomly generated datasets do not result in any compression. Therefore, we run separate experiments to evaluate GPU offloading of decompression. Figure 6 compares the performance of image smoothing when decompression occurs in 16-core CPU versus on the GPU. We vary the average run length of the input image and always execute the image smoothing kernel in the GPU. Longer run lengths imply that data can be better compressed. Decompressing data in the GPU reduces total execution time because of lower data transfer overhead and faster decompression relative to the CPU. When the average run length is 10,000, decompression in the GPU reduces total execution time by 34% on an input array of 51200×51200.

Shared GPU. A GPU can be used in shared or exclusive access mode. For SciDB this means that in the shared GPU mode, multiple chunk processing tasks can execute concurrently. Shared GPU mode results in parallel data transfers and reduces transfer latency. However, the overheads of scheduling and context switch in the GPU can degrade performance. Figure 7 shows that for both dense matrix multiplication and K-means, performance improves till four concurrent tasks are executed in the GPU, after which performance degrades. In dense matrix multiplication the performance degrades by 4% if the GPU is shared by sixteen tasks. In contrast, for PageRank and image smoothing, the performance continues to improve as the GPU is shared by more tasks. This effect is because these applications spend only a small percentage of their total time in the GPU kernel execution, thus leaving the GPU underutilized.

Observation 3: Compute kernels are the main bottleneck in CPU execution and may take even 92% of the total execution time.

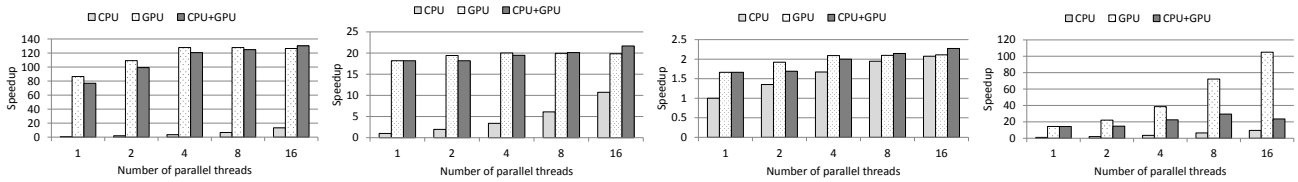
Observation 4: Data transfer, conversion, and the database itself are significant overheads and overshadow GPU kernel execution time. Database designers should use storage formats that avoid conversion, and optimize database internals.

Observation 5: GPU in shared mode can improve performance by cherry-picking number of tasks.

4.3 CPU and GPU cooperation

Finding the optimal task allocation between the CPU and the GPU is beyond the scope of this paper. Instead, we focus on three intuitive scheduling policies: (1) schedule all tasks on the CPU (policy `CPU`), (2) assign all tasks on the GPU (policy `GPU`), and (3) optimize chunk sizes dynamically depending upon whether the CPU or the GPU executes the task (policy `CPU+GPU`).

CPU case. Figure 8 shows that except for PageRank, performance of SciDB scales well as we use more CPU cores (policy `CPU`). At 16 CPU cores, dense matrix multiplication is 13× and K-means is 11×, and im-



(a) Dense matrix multiplication (b) K-means clustering (c) PageRank (d) Image smoothing

Figure 8: Performance effect of 16-core CPU and GPU cooperation. Speedups are relative to single CPU core.

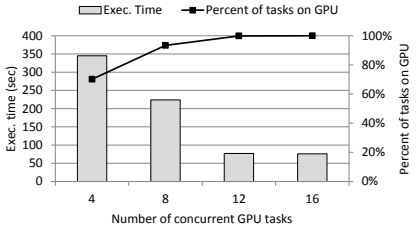


Figure 9: More tasks on GPU improve performance.

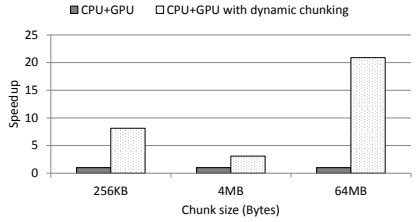


Figure 10: Benefits of dynamic chunking.

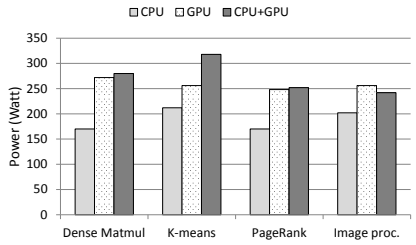


Figure 11: Power consumption. Lower is better.

age smoothing 10× faster than single core performance. PageRank has mediocre performance improvement because most of the execution time is spent inside the database to read and iterate over the dataset (Figure 5a).

GPU case. The `GPU` policy shows that scheduling all jobs on the GPU is very effective for compute intensive applications. For example, matrix multiplication is more than 127× faster than a single core system. These GPU wins remain substantial even when compared to higher core counts. At 16 CPU cores, the relative benefit of using a GPU is 1.9× for K-means, 9× for matrix multiplication, and 11× for image smoothing. PageRank, however, shows marginal performance improvement as its main kernel is not compute intensive and GPU offloading imposes additional overheads (Figure 5a).

CPU+GPU case. The `CPU+GPU` policy always performs better than 16-core CPU implementations. Compared to the `GPU` policy, this hybrid policy performs 3%-9% better in all applications except image smoothing, where it performs worse.

In image smoothing, CPU tasks take much longer (11×) than GPU tasks. As the scheduler assigns tasks to idle CPUs, they end up as stragglers and increase total execution time. Figure 9 shows that as the number of GPU tasks increase from 4 to 16, execution time decreases by 78%, which means all tasks should be scheduled on the GPU to obtain the best performance.

Figure 10 shows that dynamic chunking improves performance of matrix multiplication by 3× to 20× in the `CPU+GPU` policy. When the input chunks are very small (256KB) or large (64MB), dynamic chunking shows the maximum benefit: chunks can be merged (divided) to improve the performance of CPU or GPU tasks.

Observation 6: GPU implementations can be 127× faster than single core CPU implementations, but compared to 16-core CPU performance benefits are modest.

Observation 7: Array applications incur load imbalance due to different performance of CPU and GPU tasks. Simple scheduling policies give moderate speedups, but many applications will require more sophisticated techniques.

4.4 Power and energy efficiency

Database designers are under increasing pressure to reduce power consumption and improve energy efficiency (performance per Watt). Figure 11 shows that the `CPU` configuration consumes the least amount of power, while the `GPU` configuration consumes 20-60% more. GPU case consumes more power due to the (sometimes significant) use of CPUs for pre-processing such as data conversion. The hybrid (`CPU+GPU`) policy requires the highest power, 65% more than the CPU case.

Table 2 compares the *energy efficiency* calculated as the inverse of runtime-power product ($\frac{1}{exec\_time * power}$) and normalized to the CPU-only baseline. GPU acceleration

Policy	Dense Mat-Mul	K-means	PageRank	Image smth.
CPU	1.0	1.0	1.0	1.0
GPU	5.9	1.5	0.6	8.6
CPU+GPU	5.9	1.4	0.7	2.1

Table 2: Normalized efficiency (performance/Watt) for each configuration. Higher is better.

clearly has the energy efficiency advantage for applications such as matrix multiplication, K-means and image smoothing. Specifically, for image smoothing the GPU is 8× more energy efficient than the CPU. PageRank has lower efficiency with the GPU due to the overheads of data conversion and low utilization (e.g., 7% GPU utilization). Other than PageRank, using both the CPU and GPU is more energy efficient than using only the CPUs.

Observation 8: Multi-core CPUs have the lowest power consumption. Yet, speedup benefits makes the GPU (sometimes in cooperation with CPUs) a more energy efficient option.

5 Related Work

Previous research, on leveraging hardware accelerators for databases, has been limited to traditional relational models and query components such as indexing [2], sorting [5], and joins [7, 13]. While useful, these components are often not the dominant performance bottleneck in AODBs. Instead, we focus on array operations common in multi-dimensional data analysis. An orthogonal line of research is on GPU kernels for data analysis [14, 8, 9]. Our focus is not on creating new GPU kernels but on integrating, leveraging, and tuning the GPU kernels inside the database execution engine for array queries. Many prior work make simplifying assumptions such as limiting the dataset to completely reside in the small memory local to the accelerator (6GB for NVIDIA Tesla) or comparing performance against single threaded CPU implementations [6]. In contrast, we focus on real-world applications where these simplifications often do not hold true. Dandelion simplifies programming by compiling .NET code to different backends such as GPUs and FPGAs [10]. Observations in our paper are complementary, and help understand bottlenecks and energy efficiency in heterogeneous environments.

6 Conclusion

The Good. GPUs are a good fit for array queries. They improve performance and energy efficiency. In all our applications, using the GPU outperformed the pure multi-core CPU implementations and resulted in 20%-40% better energy efficiency (and sometimes 8× better).

The Bad. The GPU speedups are in the 10× range (instead of a desirable 100×) when compared to a multi-core CPU implementations. Additionally, it is not easy to determine the right configuration to use. For each array query, the database administrator would have to tune various software configuration parameters.

The Promising. Our observations in this paper point to the untapped potential of GPUs. In most database applications, data conversion, transfer, and database processing have a lot of overhead (more than 90% in image smoothing). By reducing these overheads we will not only boost performance but also energy efficiency. Our evaluation also shows that CPUs are an integral component for query processing. Even in the GPU mode, the CPU has a helper role, executing many parts of the query such as data partitioning, distribution, and conversion. Going forward, we expect AODBs to be optimized for the hybrid configuration.

References

[1] Monetdb with sciql. http://www.scilens.org/Goals.

[2] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. GPGPU, 2010.

[3] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. SIGMOD, 1998.

[4] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. SIGMOD, 2010.

[5] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. SIGMOD'06.

[6] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. ISPASS, 2011.

[7] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. SIGMOD, 2008.

[8] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. SIGMOD, 2010.

[9] A. Rathi, M. DeBole, W. Ge, R. T. Collins, and N. Vijaykrishnan. A gpu based implementation of center-surround distribution distance for feature extraction and matching. DATE, 2010.

[10] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. SOSP '13, pages 49–68, New York, NY, USA, 2013.

[11] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? ISCA, 2012.

[12] E. Soroush, M. Balazinska, and D. Wang. Arraystore: a storage manager for complex parallel array processing. SIGMOD, 2011.

[13] C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration for spatial selections and joins. SIGMOD, 2003.

[14] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *VLDB*, 2011.