Parallel Search On Video Cards

Tim Kaldewey, Jeff Hagen, Andrea Di Blas, Eric Sedlar
Oracle Server Technologies – Special Projects
{tim.kaldewey, jeff.hagen, andrea.di.blas, eric.sedlar}@oracle.com

Abstract

Recent approaches exploiting the massively parallel architecture of graphics processors (GPUs) to accelerate database operations have achieved intriguing results. While parallel sorting received significant attention, parallel search has not been explored. With *p-ary* search we present a novel parallel search algorithm for large-scale database index operations that scales with the number of processors and outperforms traditional thread-level parallel GPU and CPU implementations. With parallel architectures becoming omnipresent, and with searching being a fundamental functionality for many applications, we expect it to be applicable beyond the database domain. While GPUs do not appear to be ready to be adopted for general-purpose database applications yet, given their rapid development, we expect this to change in the near future. The trend towards massively parallel architectures, combining CPU and GPU processing, encourages development of parallel techniques on both architectures.

1 Introduction

Specialized hardware has never made significant inroads into the database market. Recently, however, developers have realized the inevitably parallel future of computing, and are considering alternative architectures. Current GPUs offer a much higher degree of parallelism than CPUs, and databases, with their abundance of set-oriented processing, offer many opportunities to exploit data parallelism. Driven by the large market for games and multimedia, graphics processors (GPUs) have evolved more rapidly than CPUs, and now outperform them not only in terms of processing power, but also in terms of memory performance, which is more often than computational performance the bottleneck in database applications [1].

Today's enterprise online-transaction processing (OLTP) systems rarely need to access data not in memory [2]. The growth rates of main memory size have outstripped the growth rates of structured data in the enterprise, particularly when ignoring historical data. In such cases, database performance is governed by main memory latency [3], aggravated by the ever widening gap between main memory and CPU speeds. In the 1990's the term *memory wall* was coined [4], which remains an accurate description of the situation today [5].

Workload [#Queries]	**Small** 16	**Medium** 1k	**Large** 16k
binary CPU	1.0	1.0	1.0
binary CPU SSE	1.3	1.4	1.4
binary CPU TLP	0.6	1.1	2.1
binary GPU TLP	0.3	5.1	5.1
p-ary GPU SIMD	0.5	5.4	6.7

Table 1: Measured speedup of parallel index search implementations with respect to a serial CPU implementation

The paradigm shift from bandwidth to throughput oriented, parallel computing [6] comes with new opportunities to circumvent the memory wall. Interleaving memory requests from many cores and threads theoretically allows for much higher memory throughput than optimizing an individual core could provide. Leveraging the massive parallelism of GPUs with up to 128 cores [7] we put this approach to the test.

Optimizing enterprise-class (parallel) database systems for throughput usually means exploiting thread-level parallelism (TLP), pairing each query with a single thread to avoid the cost associated with thread synchronization. The GPU's threading model exposed to the programmer suggests the same mapping, although modern GPUs architectures actually consist of multiple SIMD processors, each comprising multiple processing elements (Fig. 1). While for certain workloads this approach already outperforms similarly priced CPUs, we demonstrate that algorithms exploiting the nature of the GPU's SIMD architecture can go much further (Tab. 1).

We developed a parallel search algorithm, named *p-ary* search, with the goal to improve response time for

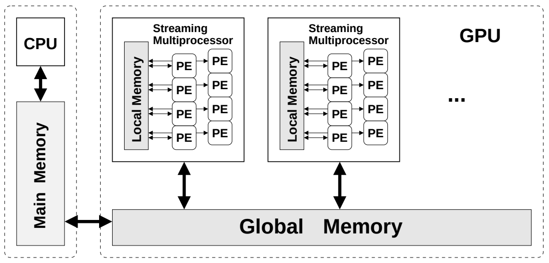

Figure 1: Architecture of an Nvidia GeForce 8 series GPU.

massively parallel architectures like the GPU, which were originally designed to improve throughout, rather than individual response time. The "p" in the name refers to the number of processors that can be synchronized within a few compute cycles, which determines convergence rate of this algorithm. As it turns out, implemented on the GPU, p-ary search outperforms conventional approaches not only in terms of response time but also throughput, despite significantly more memory accesses and theoretically lower throughput. However, small workloads yield poor performance, due to the overhead incurred to invoke GPU computation. As the GPU operates in batch mode, the often critical time-to-first-result is the time-to-completion, such that larger workloads/batches incur higher latency.

Despite promising performance results, the GPU is not ready yet to be adopted as a query co-processor for multiple reasons. First, the GPU's batch processing mode. Second, the lack of global synchronization primitives, small caches and the absence of dynamic memory allocation make it difficult to develop efficient parallel algorithms for more complex operations such as joins [8]. Third, the development environment is still in its infancies offering limited debugging and profiling capabilities.

However, GPU computing is progressing so rapidly that between the time this research was conducted and its publication some of these issues are already addressed in the latest hardware and software generation. For example, global synchronization and asynchronous communication have been added to the hardware feature list. While emerging parallel programming APIs like OpenCL [9] already blur the frontiers between CPU and GPU computing, future architectures like Larrabee [10] integrate both.

2 GPGPU

At the beginning of the computer graphics era, the CPU was in charge of all graphics operations. Progressively, more and more complex operations were offloaded to the GPU. When, thanks to their massively parallel architecture, GPUs started becoming more powerful than the CPU itrself, many programmers began exploring the use of GPU for non-graphics computations, a practice re-

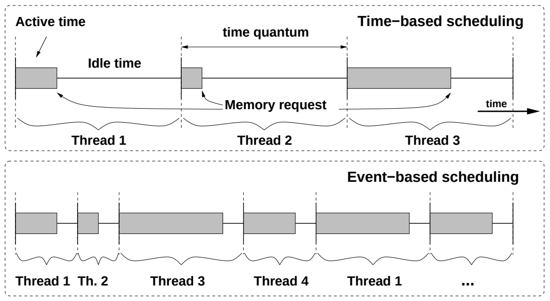

Figure 2: Comparison of scheduling techniques. *Event-based scheduling* on the GPU maximizes processor utilization by suspending threads as soon as they issue a memory request, without waiting for a time *time quantum* to expire, as on the CPU.

ferred to as *General-Purpose Graphics Processing Unit*, or GPGPU. However, programming the GPU had to be done by using standard graphics APIs, such as OpenGL or DirectX. Even when using it for general-purpose computations, the developer had to map data and variables to graphics objects, using single-precision floating-point values, the only data type available on the GPU. Algorithms had to be expressed in terms of geometry and color transformations, and to actually perform a computation required pretending to draw a scene. This task was usually rather challenging even for simple applications, due to the rigid limitations in terms of functionality, data types, and memory access.

Nvidia's *Compute Unified Device Architecture* (CUDA), an extension to the C programming language that allows programming the GPU directly, was a major leap ahead [11]. At the top level, a CUDA application consists of two parts: a serial program running on the CPU and a parallel part, called a *kernel*, running on the GPU.

The kernel is organized as a number of *blocks* of *threads*, with one block running all its own threads to completion on one of the several *streaming multiprocessors*, SMs, available. When the number of blocks as defined by the programmer exceeds the number of physical multiprocessors, blocks are queued automatically. Each SM has eight processing elements, PEs (Fig. 1). PEs in the same streaming multiprocessor execute the same instruction at the same time in *Single Instruction-Multiple Data*, SIMD, mode [12].

To optimize SM utilization, within a block the GPU groups threads following the same code path into so called *warps* for SIMD-parallel execution. Due to this this mechanism, nVidia calls its GPU architecture *Single Instruction-Multiple Threads*(SIMT). Threads running on the same SM share a set of registers as well as a low-latency *shared memory* located on the processor chip. This shared memory is small (16 KB on our G80) but about 100× faster than the larger *global* mem-

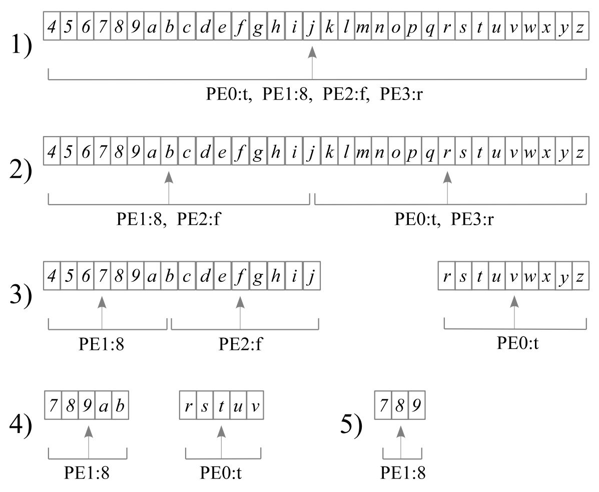

Figure 3: Four PEs independently performing a binary search for different keys on the same data range in five steps.

ory on the GPU board. A careful memory access strategy is even more important on the GPU than it is on the CPU because caching on the GPU is minimal and mainly the programmer's responsibility.

To compensate for the small local memories and caches GPUs employ massive multithreading to effectively hide memory latency. The scheduler within an SM decides for each cycle which group of threads (warp) to run, such that warps with threads accessing memory can be suspended at no cost until the requested data is available. The seamless multithreading is made possible by thousands of register in each SM, such that each thread keeps its variables in registers and context switching is free. Effectively this approach implements what we would naively describe as event-based scheduling (Fig. 2) and benefits large, latency-bound workloads.

On the other hand, CPUs employ larger caches (4 MB on our Q6700) but rely on a single set of registers, such that context switches require preserving the state of execution of the current thread before loading the next. As context switching is expensive and schedulers are implemented in software, CPU scheduling is based on time quanta such that in case of a cache miss a thread sits idle until the memory request returns or its time quantum expires.

These characteristics make the GPU an interesting platform for parallel database processing.

3 GPU Parallel Search

Video cards have been explored as coprocessors for a variety of non-graphics related applications [13] including database operations [14]. Sorting on the GPU [15, 16] including very large data sets that require multi-pass sorting [17], used variants of Batcher's bitonic sorting networks [18]. While geospatial databases, whose data sets are similar to graphics data were the first to adopt

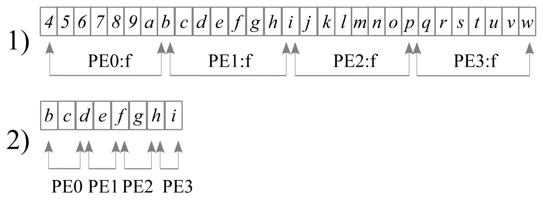

Figure 4: Four PEs jointly performing a search based on domain decomposition in 2 steps. At each step all PEs are reassigned within a sub-range.

GPU's for more complex operations like join [19], this has only been considered recently for general-purpose databases [8]. GPGPU research prior to 2007, before the release of nVidia's CUDA [11], required the use of graphics specific APIs that only supported floating-point numbers. However, fast searching — which is fundamental even beyond database applications — has not been explored for general purpose data on the GPU yet.

The obvious way of implementing search on the GPU is to exploit data parallelism, omnipresent in large-scale database applications, handling thousands of queries simultaneously. Multiple threads run the same serial algorithm on different problem instances, which is no different than CPU multi-threading where each select operation is paired with a single thread. While this approach does not improve response time for a single problem instance, it returns multiple results at the same time while requiring only minimal synchronization. To reduce response time for a single problem instance, we suggest to explore functional parallelism following the divide-and-conquer principle, assigning different sub-problems to different PEs. The efficiency of this approach is only limited by the amount of communication and the synchronization overhead imposed by the architecture. Since synchronization within a SM is relatively fast, this appears to be a promising avenue.

Parallel Binary Search. For a data-parallel search implementation we selected multithreaded serial binary search, which has been used in the past for graphics processing [20]. The worst-case run time of binary search on n elements is $log_2 n$, but a multithreaded parallel implementation using p PEs can perform p searches simultaneously. Although some searches might finish earlier, in SIMD systems the other PEs must wait to start the next search until all are finished. A workload with a number of searches smaller than the number of PEs renders the remaining PEs idle and results in inefficient resource utilization

Figure 3 shows a parallel binary search of four keys, one per PE, in the same search space — the characters from '4' to 'z'. In our example PE0, PE2, and PE3 found their data quickly and have to idle until PE1 finishes. The

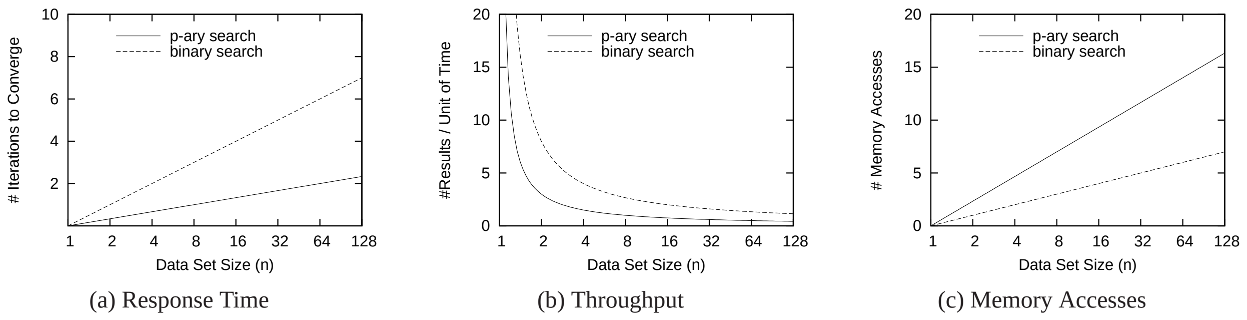

Figure 5: Theoretical comparison of p-ary and parallel binary search for $p = 8$, the number of PE's in a SM.

larger the number of PEs in a SM the more likely it is that p searches require worst-case execution time.

P-ary Search. A domain decomposition strategy applied to search has all PEs within an SM searching for the same key value, each in a disjoint subset of the initial search range (Fig. 4). At each step all PEs compare their key with the two boundary values of their subset. Afterwards, all PEs are reassigned to new disjoint subsets within the subset identified as containing the search key value. In case more than one PE reports finding the key value, it is irrelevant which one delivers the result since all are correct.

This approach reduces the search range by $1/p$ at each iteration, yielding a worst-case execution time of $log_p n$. The response time of this algorithm is significantly lower than the previous one, but it delivers only one result for each run instead of p. However, it has higher efficiency since PEs never idle and the synchronization overhead is minimal on SIMD architectures. Neighboring PEs can share a boundary key, to reduce the number of global memory accesses from $2\ p$ to $p+1$ for each iteration. This can be further reduced to $p-1$ by re-using the result's boundary keys from the previous iteration or setting the lower bound to $-\infty$ and the upper one to $+\infty$.

Theoretical Considerations. A brief theoretical evaluation of p-ary search regarding throughput, rate of convergence, and the number of memory accesses exhibits the tradeoffs necessary to achieve better convergence and thus response time (Fig. 5a). Figure 5b shows that p-ary search achieves significantly lower throughput than binary search, requiring $log_p(n)$ time for one result vs $log_2(n)$ time for p results. Although p-ary search requires the same number of memory accesses per iteration, p, it yields only 1 result as opposed to p. The faster rate of convergence cannot compensate for the more than $2\times$ larger (Fig. 5c) amount of memory accesses.

However, in practice p-ary search outperforms multithreaded binary search by 30% on the same architecture (Sec. 4). The reasons why p-ary search is still able to achieve better throughput are manyfold.

First, memory requests by multiple PEs can be served in parallel due the GPUs wide memory buses (384-bit for our G80), as long as they do not conflict. Without caching, on the GPU conflicting memory requests are serialized. P-ary search produces converging memory strides, with a stride length determined by the subset searched divided by the number of PEs. Memory conflicts can only occur in the last iteration if the remaining subset contains less entries than processors.

Second, multithreaded binary search will produce many memory conflicts as all PEs start with the same pivot element, before they diverge. The first iteration is guaranteed to produce number of PEs, p, conflicts resulting in p serialized memory accesses. While the probability decreases from iteration to iteration, binary search is guaranteed to produce a minimum of $p\ log_2 p$ memory conflicts, while p-ary search $p-1$ at most.

Third, p-ary search has a smaller footprint in terms of register and local memory usage. This allows to run more instances (threads) of p-ary search on a streaming multiprocessor than of binary search. Therefore, performance gains over binary search only become apparent for larger workloads (Fig. 6).

4 Experimental Evaluation

We analyze the performance of our parallel search algorithm applied to database index queries, with respect to throughput, scalability, and response time. All experiments were conducted on an x86 system with a 2.66 GHz Intel Core2-Quad Q6700 CPU, DDR2-1066 5-5-5-15 RAM, and two 1.35 GHz nVidia GeForce 8800GTX GPUs with 768 MB DDR3 RAM each, running Oracle Enterprise Linux 5, kernel 2.6.18, CUDA 1.0 and nVidia display drivers 100.14.11.

The data set was designed to resemble database structures, with a maximum size limited by the video card memory, which also has to accommodate queries, results and the search application. The data consists of randomly-generated 15-character null-terminated ASCII strings organized as in a database table. For the evaluation of our index search implementations we use a sorted column with 36 million unique entries, approximately

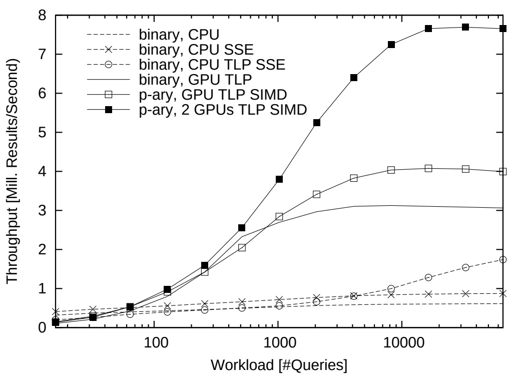

Figure 6: Absolute performance of index search implementations exploiting thread level and SIMD parallelism on the CPU and the GPU with respect to workload.

550 MB in size, resembling an index column of a large database table.

The following experiments resemble a busy OLTP database server with a varying number of requests in its input queues. For the GPU implementation we permanently store the data set in video memory and for the CPU implementation in main memory, as in-memory databases would. Transferring our 550 MB data set from main memory to GPU memory takes 400 ms, but is only required at startup. An efficient GPU accelerated database would require either the memory being shared between CPU and GPU, or updates taking place directly in the GPU memory, such that transfer time is irrelevant.

Throughput. Figure 6 shows the throughput achieved by different search implementations dependent on workload conditions. Starting with a serial CPU implementation of binary search as the baseline, there are significant performance gains from exploiting the CPU's SSE vector unit for all workloads, while multithreading only yields performance gains for larger workloads. The same applies to GPU implementations which in general require larger batches in order to achieve good performance. However, handling large amounts of queries simultaneously is nothing unusual for large-scale database servers. Similar to the CPU, the GPU benefits from exploring SIMD parallelism for all workloads, such that index operations using p-ary search yield up to 30% better performance.

As the low performance of GPU search implementations on small workloads in the previous experiment already indicates, offloading work to the GPU involves significant startup overhead. Thus we measure the time that each step in the offloading process requires, i.e. API launch time, processing time and copying queries to and results from the GPU (Fig. 7). For a detailed analysis of execution time of database functions on the CPU we

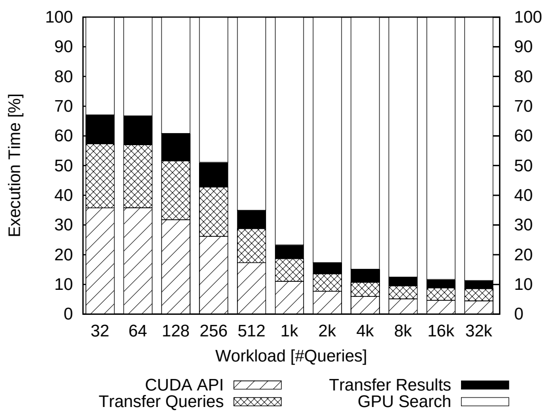

Figure 7: Timing breakdown for offloading batches of select queries(searches) to the GPU.

would like to refer to the work by Ailamaki et al. [3].

Timing Breakdown. We measure the execution time for each step by running exactly the same batch of queries multiple times, each time adding another step of the offloading process. We obtain the time required for a step by computing the difference to the previous run. For example, the API launch time is determined by executing an empty program. The time for transferring the batch of queries to the GPU is determined by subtracting the time required to launch an empty program from the time required for launching the program and copying the queries to the video card, and so on.

As expected, Figure 7 shows that for small workloads the total time is dominated by the cost for offloading the processing to the GPU, e.g. for a workload of 32 queries approximately 70% of the time is spent launching the API and copying data back and forth. As the workload size increases this overhead decreases and execution time becomes the dominant factor. This also marks when offloading to the GPU becomes efficient.

Response Time. Efficiency is not always the only factor that needs to be considered when offloading work to the video card. For many applications like web services for example, time-to-first result or response time needs to be considered when deciding wether to offload queries to the video card. Given the GPU's batch programming model the time-to-first-result or response time equals time-to-completion. In Figure 8 we measured the sustained index search throughput given a maximum response time, which would in turn determine the batch size. In order to compare these results to CPU implementations we did not allow the CPU to return results early. For response times of 2 ms and above, the throughput of two GPUs reaches its peak performance of 7.5 million index searches per second. Even with the GPU operating

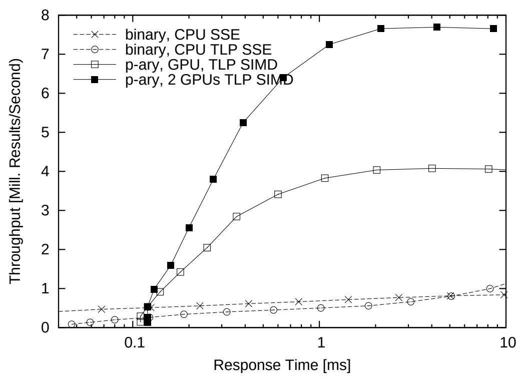

Figure 8: Achievable throughput of index operations with respect to response time requirements

in batch mode it can be used efficiently as a co-processor in a system with response time requirements of 0.2 ms and above.

5 Conclusions and Future Work

We strongly believe that exploring parallelism beyond traditional multi-threading and even beyond the threading model on the GPU's SIMT architecture (Sec. 2) has great potential. As we have demonstrated with p-ary search there can be significant performance gains developing truly parallel algorithms. Theoretically, p-ary search converges much faster than binary search, but on the other hand has significantly lower throughput and requires more memory accesses. In practice, we found that p-ary search outperforms binary search on the GPU in terms of throughput, despite nearly three times as many memory accesses. For database systems, besides throughput, response time and time-to-first-result are crucial, and p-ary search on the GPU improves all three.

As p-ary search and our research analyzing memory performance [21] demonstrate, parallel memory accesses are a way to scale not only memory latency but also throughput. With rapidly increasing main memory sizes, it is expected that soon even traditionally I/O bound database applications like Business Intelligence associated with Data Warehousing can leverage in-memory analytics [22]. With more and more applications becoming memory bound, overcoming the memory wall [4, 5] is imperative.

We will continue our research on parallel algorithms, leveraging parallelism to improve performance of memory-bound applications, using in-memory databases as an example. To validate this concept, we are porting p-ary search to other parallel architectures, including Intel Nehalem, Sun Niagara, and IBM Cell. While implementing p-ary search to multi-core architectures is straightforward, it is also possible to map p-ary search to vector architectures, for example x86 SSE which is going to double in width in the next generation [23]. The rapid improvements of GPU programmability and Efforts like Intel Larrabee [10] indicate a fusion of GPU and CPU architectures such that algorithms will be applicable to future architectures. We focus on algorithms designed to scale with core count, such that even in the current era of throughput-oriented computing [6], response time will improve as well.

References

[1] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB '99*.

[2] T. T. Team, "High-performance and scalability through application tier,in-memory data management," in *VLDB'00*.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *VLDB'99*.

[4] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[5] S. A. McKee, "Reflections on the memory wall," in *Conf. on Computing Frontiers (CF'04)*.

[6] C. Hampel, "Terabyte bandwidth initiative — architectural considerations for next-generation memory systems," in *Hot Chips 20*, 2008.

[7] nVidia, "GeForce 8 Series," 2007, http://www.nvidia.com/page/geforce8.html.

[8] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD '08*.

[9] A. Munshi, "OpenCL — parallel computing on the GPU and CPU," in *SIGGRAPH '08*.

[10] D. Carmean, "Larrabee: A many-core x86 architecture for visual computing," in *Hot Chips 20*, 2008.

[11] nVidia, "NVIDIA CUDA homepage," 2008, http://developer.nvidia.com/object/cuda.html.

[12] M. J. Flynn, "Very high-speed computing systems," in *Proc. of the IEEE*, vol. 54(12), 1966, pp. 1901–1909.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[14] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGMOD '04*.

[15] D. Callele, E. Neufeld, and K. DeLathouwer, "Sorting on a GPU," 2003, http://www.cs.usask.ca/faculty/callele/gpusort/gpusort.html.

[16] A. Greß and G. Zachmann, "GPU-ABiSort: Optimal parallel sorting on stream architectures," in *IPDPS'06*.

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics co-processor sorting for large database management," in *SIGMOD '06*.

[18] K. E. Batcher, "Sorting networks and their applications," in *AFIPS Spring Joint Computing Conf.*, 1968.

[19] N. Bandi, C. Sun, A. E. Abbadi, and D. Agrawal, "Hardware acceleration in commercial databases: A case study of spatial operations," in *VLDB '04*.

[20] I. Buck and T. Purcell, "A Toolkit for Computation on GPUs," in *GPU Gems, Chapter 37*, R. Fernando, Ed. New York: Addison Wesley, 2004, pp. 621–636.

[21] T. Kaldewey, A. D. Blas, J. Hagen, E. Sedlar, and S. A. Brandt, "Memory matters," in *RTSS'06*.

[22] K. Schlegel, "Emerging technologies will drive self-service business intelligence," February 2008, Gartner Report. ID Number G00152770.

[23] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," 2008, Intel White Paper.