A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors

Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, Dinesh Manocha
University of North Carolina at Chapel Hill
{naga,nikunj,henson,tuft,dm}@cs.unc.edu
http://gamma.cs.unc.edu/GPUSORT

Abstract

We present a fast sorting algorithm using graphics processors (GPUs) that adapts well to database and data mining applications. Our algorithm uses texture mapping and blending functionalities of GPUs to implement an efficient bitonic sorting network. We take into account the communication bandwidth overhead to the video memory on the GPUs and reduce the memory bandwidth requirements. We also present strategies to exploit the tile-based computational model of GPUs. Our new algorithm has a memory-efficient data access pattern and we describe an efficient instruction dispatch mechanism to improve the overall sorting performance. We have used our sorting algorithm to accelerate join-based queries and stream mining algorithms. Our results indicate up to an order of magnitude improvement over prior CPU-based and GPU-based sorting algorithms.

1 Introduction

Sorting is an integral component of most database management systems (DBMSs) and data stream management systems (DSMSs). The performance of several DBMS and DSMS queries is often dominated by the cost of the sorting algorithm. Therefore, many techniques have been proposed to analyze and improve the performance of sorting algorithms. Recent studies have indicated that sorting can be both computation-intensive as well as memory-intensive. These investigations on the processor and memory behaviors indicate that there can be substantial overhead in database and data mining applications due to memory stalls on account of data cache misses, branch mispredictions, and resource stalls due to instruction dependencies. Many algorithms have been proposed to improve the performance, including data parallel algorithms [40], cache conscious data structures and algorithms [10, 36, 38], instruction buffering algorithms [41] and better data storage models [3]. These techniques have been extensively studied for CPU-based algorithms.

Recently, database researchers have been exploiting the computational capabilities of graphics processors to accelerate database queries and redesign the query processing engine [8, 18, 19, 39]. These include typical database operations such as conjunctive selections, aggregates and semi-linear queries, stream-based frequency and quantile estimation operations,and spatial database operations. These algorithms utilize the high memory bandwidth, the inherent parallelism and vector processing functionalities of GPUs to execute the queries efficiently. Furthermore, GPUs are becoming increasingly programmable and have also been used for sorting [19, 24, 35].

The GPU-based sorting algorithms use the basic functionalities of GPUs and can outperform popular CPU-based algorithms such as quicksort [19]. Most work on GPU-based sorting focused on using different features and capabilities of GPUs to design better sorting algorithms. Though efficient, current GPU-based algorithms tend to be memory-limited. As compared to CPU-based sorting algorithms, there is relatively little or no work on optimizing the memory performance of GPU-based sorting algorithms for database applications.

The GPUs have a high memory bandwidth. However, the performance of GPU-based sorting algorithms is dominated by the latency in memory accesses. In order to reduce memory latencies, current GPUs use L1 and L2 caches internally. These caches are optimized for rasterizing primitives, and are usually designed based on a tile-based architecture of GPUs. Furthermore, there can be restrictions within the memory access model of graphics processors. Therefore, cache-conscious optimizations designed for CPUs are not directly applicable to GPUs.

Main Results: In this paper, we present a novel GPU-based sorting algorithm that offers improved performance as compared to prior GPU-based algorithms. Our algorithm is based on the bitonic-sorting network and uses the inherent parallelism and vector processing functionalities

of GPUs. We use the texture mapping, blending or programmable functionalities of the GPUs to efficiently implement the sorting network. In practice, our algorithm is memory-bound and we present different techniques to improve its memory efficiency. Our overall algorithm has a better memory access pattern and locality in data accesses and significantly improves the cache efficiency and the overall performance. We have used our fast sorting algorithm to accelerate the computation of equi-join and non-equi-join queries in databases, and numerical statistic queries on data streams.

We have implemented our algorithms on a PC with a 3.4 GHz Pentium IV CPU and with an NVIDIA GeForce 6800 Ultra GPU. We have used our algorithm on databases consisting of up to a million values and data streams consisting of more than 100 million values. Our results indicate a factor of $4 - 10$ times improvement over prior CPU-based and GPU-based algorithms. In practice, our cache-efficient optimizations result in more than 30% improvement in the computational performance.

Organization: The rest of the paper is organized in the following manner. We briefly survey related work on sorting and database performance in Section 2. Section 3 gives an overview of GPUs and memory layout. We present our sorting algorithm in Section 4 and highlight its application to database and data streaming operations in Section 5. We analyze the performance of our algorithm in Section 6.

2 Related Work

In this section, we give a brief overview of prior work on sorting algorithms and acceleration of database operations.

2.1 Sorting

Sorting is a well studied problem in the theory of algorithms [25]. In fact, optimized implementations of some algorithms such as Quicksort are widely available. These include optimized implementations available as part of standard compilers such as Intel C++ compiler and Microsoft Visual C++ 6.0 compiler. The implementation of Quicksort in the Intel compiler has been optimized using Hyper-Threaded technology. More details on the implementation of Quicksort are given here[1]. In the database literature, many fast algorithms have also been designed for transaction processing and disk to disk sorting [1]. However, the performance of sorting algorithms on conventional CPUs is governed by cache misses [26] and instruction dependencies [40].

In terms of using GPUs for sorting, Purcell et al. [35] described an implementation of bitonic merge sort on the GPUs. The bitonic sort is implemented as a fragment program and each stage of the sorting algorithm is performed as one rendering pass. Kipfer et al. [24] presented an improved bitonic sort routine that achieves a performance gain by minimizing the number of instructions in a fragment program and the number of texture operations, but the algorithm still requires a number of instructions. More recently, Govindaraju et al. [19] presented an improved sorting algorithm using texture mapping and blending operations, which outperforms earlier GPU-based algorithms.

2.2 Hardware Accelerated Database Operations

Many algorithms have been proposed to improve the performance of database operations using multi-level memory hierarchies that include disks, main memories, and several level of processor caches. A recent survey on these techniques is given by Ailamaki [2]. Over the last few years, database architectures have moved from disk-based systems to main-memory systems and the resulting applications tend to be either computation-bound or memory-bound. In particular, memory stalls due to cache misses can lead to increased query times [4, 28]. There is considerable recent work on redesigning database and data mining algorithms to make full use of hardware resources and minimize the memory stalls and branch mispredictions [3, 10, 27, 31, 36, 37, 38, 40].

Many acceleration techniques have been proposed to improve the performance of database operations. These include use of vector processors to accelerate the execution of relational database operations including selection, projection, and join [31] and SIMD implementations of scans, aggregates, indexed searches and joins [40]. Recently GPUs have been used to perform database and data mining computations including spatial selection and join [39], spatial database operations within Oracle 9I DBMS [8], predicates, boolean combinations and aggregates [18], and stream data mining [19].

3 Overview

In this section, we provide a brief overview of our cache-efficient sorting algorithm. We describe the essential capabilities of GPUs that are used to obtain improved sorting performance. We first give an overview of GPUs and describe some of their architectural features that influence the sorting performance. These include the texture access patterns used for comparison between data elements, data parallelism, and instruction dispatch.

3.1 Background

The performance of some database and data mining applications is largely dependent upon sorting. In these appli-

[1] http://www.intel.com/cd/ids/developer/asmona/eng/20372.htm?prn=Y

cations, sorting is often considered an expensive operation. In particular, the performance on modern CPUs is mainly governed by the sizes of the L1 and L2 caches. These cache sizes are typically small (of the order of a few hundreds KB or a few MB) and studies indicate that CPU-based algorithms such as Quicksort can incur larger cache penalty misses [26, 19]. As a result, the CPU-based sorting algorithms do not achieve their peak computational performance [26]. Moreover, conditional branches in sorting algorithms often lead to pipeline stalls and can be expensive [40].

In contrast, GPUs are designed using SIMD vector processing units that can access a high bandwidth video memory. These programmable SIMD units offer a better compute-to-memory performance. For example, a high-end GPU such as NVIDIA GeForce 6800 Ultra can perform up to 64 comparisons in one GPU clock cycle and can achieve a peak memory bandwidth of 35.2 GBps. As a result, GPU-based sorting algorithms such as periodic balanced sorting network algorithm (PBSN) [19] on a high-end GPU can outperform well-optimized CPU-based sorting algorithms (e.g. Quicksort) running on a high-end CPU.

The performance of GPU-based sorting algorithms is primarily limited by the memory bandwidth. Similar to CPUs, current GPUs also have L1 and L2 texture caches and use memory layout schemes to improve the memory bandwidth. A recent study on the memory performance in fast matrix-matrix multiplication algorithms using GPUs [15] indicates a poor performance on the use of CPU-based blocking algorithms to improve the cache performance. There is relatively little work on memory access patterns of GPU-based sorting algorithms as compared to the CPU-based algorithms. In this paper, we present a theoretical and experimental study to demonstrate that the sorting performance on GPUs can be significantly improved by designing cache-efficient algorithms. In the rest of this section, we provide a brief overview of the capabilities of the GPUs used to implement a fast sorting algorithm. After that, we present the computational model used for rasterizing primitives using GPUs and the memory organization on the GPUs.

3.2 Graphics Processors

The graphics processor is designed to perform vector computations on input data represented as 2D images or textures. Each element of a texture is composed of four color components, and each component can store one floating point value. We represent the input data in 2D textures and perform comparison operations to sort the data elements. To access the data on the GPUs and perform comparisons between them, we use two main features - texture mapping and color blending operations. These two features operate on the 2D images or textures. Texture mapping is used to map the input color from the texture to each pixel on the screen. The mapping is specified by rasterizing a quadrilateral that cover the pixels on the screen and each vertex of the quadrilateral is associated with texture co-ordinates. The texture mapping hardware performs a bi-linear interpolation of the texture co-ordinates to compute the mapped co-ordinates for each pixel that is rasterized. A 2D lookup is performed on the active texture or image, and the input color is assigned to the pixel. The blending hardware is used to compare the input color with the pixel color in the frame buffer. For a detailed overview of these functionalities, refer to [29].

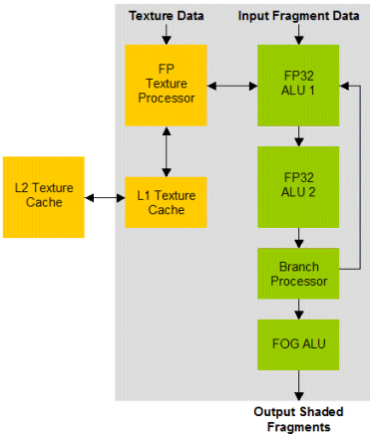

Figure 1. *Texture caches on a commodity GPU: NVIDIA GeForce FX 6800 : It has 16 programmable fragment processors. The fragment processors have a high memory bandwidth interface to the video memory. The GPU has a core clock of 400 MHz and a memory clock of 1.2 GHz, and can achieve a peak memory bandwidth of 35.2 GBps. Each fragment processor has access to a local L1 texture cache and multiple fragment processors share accesses to an L2 texture cache.*

3.2.1 Texture Memory Layout

We now present a broad overview of the texture memory layout and the tile-based computational model for rasterizing primitives on the GPUs. These concepts are used in improving the runtime performance of our sorting algorithm.

The texture memory layout and access on graphics processors is highly optimized for rendering applications. Hakura and Gupta [21] present the design and analysis of the texture memory architecture on GPUs. We briefly overview some of the general principles for efficient texture memory accesses on GPUs.

The video memory is a DRAM memory system and can have high latency for data accesses. In order to achieve high computational throughput, each fragment processor is associated with a local L1 texture cache. Moreover, a L2 texture cache is shared among multiple fragment processors as shown in Fig. 1. The L1 and L2 texture caches are static memories and can provide fast texture accesses

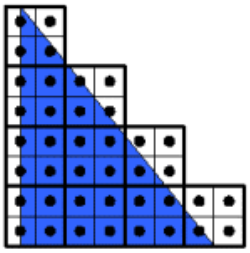

Figure 2. *Tiling on a commodity GPU: NVIDIA GeForce FX 6800: Each input triangle is decomposed into many small 2×2 pixel tiles. Each tile is processed by a single quad fragment processor pipeline, and four pixel tiles can be processed simultaneously. Note that the triangle does not entirely cover all the pixel tiles. In the partially covered pixel tiles, the fragment processors are active only on the covered pixels. The efficiency of a tile-based architecture is therefore dependent upon the size of the rasterized primitives.*

and reduce the latencies. Furthermore, the texture memory is internally represented as 2D regions known as *blocks* or *tiles*. As graphics applications exhibit a high amount of spatial coherence, a block transfer of cache lines between the cache and the video memory usually leads to the most efficient memory bus utilization. The blocked representation exploits the spatial locality during texture accesses. Therefore, data representations that access texels close to each other can improve the performance. Pre-fetching techniques are also applied to further reduce the latency of data accesses [22].

3.2.2 Rasterization

Graphics processors often traverse the rasterized pixels of a triangle in a tiled order to improve the coherence [23, 30, 32, 5]. In this scheme, the screen is statically decomposed into small rectangular regions known as tiles and all the pixels within a tile are traversed before rasterizing the next tile. In current GPUs such as an NVIDIA GeForce FX 6800, the triangles are first decomposed into 2×2 pixel tiles. Each tile is then processed by a quad pipeline or a unit with four SIMD processors. The GPU consists of four such quad pipelines and at any instance, four pixel tiles can be processed simultaneously [33]. Fig. 2 illustrates the tile decomposition of a triangle. Rasterization based on pixel tile decomposition has the benefit of efficient memory accesses as it requires a small working set from the texture memory and can fit well in the texture cache. However, tiling is not efficient for rendering thin primitives as some of the processors in the 2×2 pixel tiles could be inactive and thereby affect the overall performance.

4 Sorting on GPUs

In this section, we present our sorting algorithm and highlight its advantages over prior GPU-based sorting algorithms. We also analyze the computational complexity and the memory access bottlenecks of our algorithm. Next, we analyze the memory access patterns on GPUs and present techniques to improve the performance. We also describe efficient instruction dispatch mechanisms used to improve the computational throughput.

4.1 Sorting Networks

Current graphics processors do not support scatter operations i.e., the fragment processor cannot write to arbitrary memory locations. This restriction avoids a write-after-a-read hazard between multiple fragment processors accessing the same memory location. Therefore, most sorting algorithms such as Quicksort cannot be efficiently implemented on GPUs. On the other hand, sorting network algorithms are a class of algorithms that map well to the GPUs. Govindaraju et al. [19] provide a detailed overview of sorting network algorithms and present a fast implementation of a periodic balanced sorting network (PBSN) algorithm. We briefly summarize their results.

A sorting network algorithm proceeds in multiple steps. In each step, a comparator mapping is created at each pixel on the screen and the color of the pixel is compared against exactly one other pixel on the screen. The minimum is stored in one of the pixels and the maximum is stored in the other. The comparison operations are performed as vector operations using the blending functionality of the GPU and the mapping is implemented using the texture mapping hardware. In each step, we update the portions of the screen where the minimum values are stored by setting the blend function to output the minimum value. We then draw quads with appropriate texture coordinates on these portions of the screen. We update the portions of screen where maximum values are stored in a similar manner. For more details, refer to [19]. Alternately, we can also simulate the blending functionality using two single instruction fragment programs for computing the maximum and minimum values respectively.

4.1.1 Our Algorithm

We design an optimal sorting network that is more efficient than PBSN. Our algorithm is based on the bitonic sorting algorithm [9]. The bitonic sorting algorithm consists of a sorting network that can sort bitonic sequences. A bitonic sequence is a merger of two monotonic sequences. The bitonic sorting algorithm proceeds in multiple stages, similar to the PBSN. The overall algorithm has $\log n$ stages, and for each stage i, we perform i steps. In particular, stage i is used to merge two bitonic sequences of sizes 2^{i-1} and generate a new bitonic sequence of length 2^i. Our algorithm

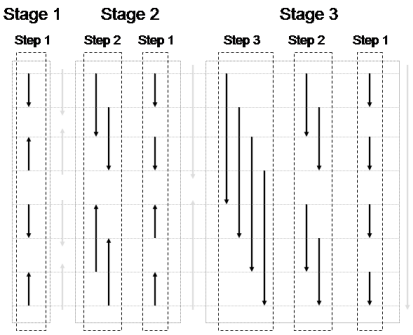

Figure 3. *A bitonic sorting network for 8 data values: Each arrow denotes a comparator which stores the smaller of the two values at the location corresponding to the tail and the larger value is stored at the head. The algorithm on n input values proceeds in $\log n$ stages, and the algorithm processes the stages from $1 to \log n$. Furthermore, each stage proceeds in multiple steps and an intermediate stage i performs i steps. Within every step, each data value is compared against another data value at a constant distance. Furthermore, the distance remains constant within each step and in step i it is equal to 2^i. However, the direction of accesses changes based on the stage. In stage j, the direction of access reverses every 2^{j-1} data values as shown in the figure. The reversal increases the distance between successively accessed data values and can lead to poor memory access patterns.*

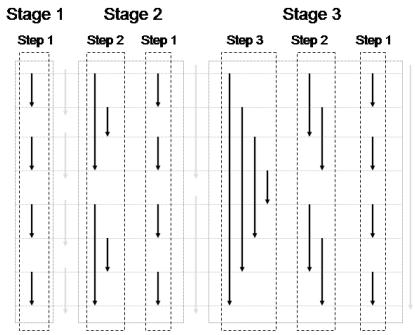

Figure 4. *Our improved bitonic sort network: The modified network is very similar to the one shown in Fig. 3. It proceeds in the same number of stages and steps. Moreover, it processes the stages and steps in the same order. Furthermore, for each element the distance and the direction of accessing its corresponding element is the same in all the steps except the first step in each stage. In the first step, one of the bitonic sequences is reversed and the accesses are performed similar to the first step of PBSN [19]. Our improved algorithm is simple to implement, has better locality in accesses and is more cache-efficient.*

repeatedly applies a sorting network to each of the stages $i = 1, 2, \ldots, \log n$. At the end of the $\log n^{\text{th}}$ stage, we obtain a sorted sequence of length n.

Fig. 3 illustrates the bitonic sorting network on 8 data values. Each arrow between the two elements indicates a two-element comparator. The maximum is stored in the element pointed by the arrow head and the minimum is stored in the other. For any given stage i and step j of the algorithm, every element is compared against an element at a distance of 2^{j-1}. However, every 2^{i-1} elements, the direction of data accesses that an element compares against is reversed. On graphics processors, the memory is conceptually represented in 2D textures of width W and height H. We map the 1D input data into corresponding 2D values using texture mapping. An element (x,y) in the texture corresponds to an element at the location $(y * W + x)$ in a 1D array. In each step, we update all the pixels on the screen that correspond to the regions where the minimum values are stored. These regions are either column-aligned quads or row-aligned quads (as shown in Fig. 5). Moreover, we ensure that we reverse the directions every 2^{i-1} elements by reversing the texture co-ordinates of the vertices of these quads. The overall algorithm requires $(5\ n\ \log n\ \frac{(\log n+1)}{2})$ memory accesses and $(n\ \log n\ \frac{(\log n+1)}{2})$ floating point comparisons.

The comparison operations are fast, and each blending-based comparison operation requires a single GPU clock cycle [34]. For large input arrays (in the order of millions), the memory access bandwidth is substantially large (tens of giga bytes per second) and the algorithm is memory-limited, as opposed to compute-limited. To utilize the high computational power of graphics processors, we improve the performance of algorithm using three properties:

- **Memory Access Pattern:** Memory access patterns of an algorithm can significantly affect the memory bandwidth. To understand its influence, we now consider the pattern of memory accesses within stage i and step j of the bitonic sorting algorithm. In particular, we examine the distance between two consecutively rendered quads as they affect the locality of data accesses. This distance is equal to 2^{j-1} if the direction of accesses is not reversed. However, when the direction is reversed, the distance between two consecutively rendered quads becomes 2^j. The increase in the distance between the quads can significantly decrease the overall performance. We modify our algorithm to improve the locality of accesses by effectively reducing the distance between the rendered quads, and thereby improve the performance.

- **Cache-Efficiency:** The memory accesses are block-based, and therefore, the size of two-consecutively rendered quads can affect the number of cache misses. We observe that in our sorting algorithm, based on the internal block sizes in the video memory, there can be multiple references to the data values within the same block fetched from the texture memory. The fre-

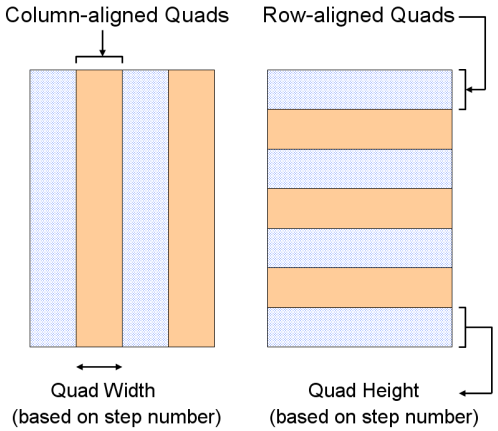

Figure 5. *Mapping of Bitonic Sort on GPUs: We use the texture mapping and blending functionalities for mapping the regions on the textures to compute the minimum values or the maximum values. These regions that correspond to either the minimum values or the maximum values are row-aligned or column-aligned quads. The width or the height of the quads is dependent upon the step number, and denotes the distance between accessed elements for performing comparisons. Larger steps require rasterization of quads with larger width and lead to more cache misses. Lower steps require the rasterization of thin quads and may not be efficient.*

quency of such references *increases* at the lower steps of the algorithm. Moreover, these lower steps are performed more frequently by the algorithm. For example, step 1 is performed $\log n$ times whereas step $\log n$ is performed only once. It is important for the algorithm to improve and reuse memory accesses based on the block accesses. Otherwise, the number of cache misses can increase. To illustrate this issue, let us consider the case of memory accesses in a column-aligned quads. If the height of the column is too large for the cache size, then due to the block access model of texture memories, the cache will contain data values at the bottom of the column as opposed to the top of the column. Therefore, the memory references at the top of the next consecutive column-aligned quad, will typically result in cache misses. We propose a simple tile-based sorting algorithm to improve the cache efficiency.

- **Data Layout:** The 2D texture-based data layout used on GPUs for sorting results in a number of thin column-aligned and row-aligned quads. For example, the width of a row-aligned quad in step 1 is unity and there are W such quads to render. These thin quads are not optimal for rendering as they involve large vertex processing overheads. Furthermore, as these quads occupy a portion of the pixel tiles (as described in section 3.2.2), many of the fragment processors are inactive and this affects the parallelism and the computational throughput. We use stenciling techniques to reduce the overhead of rendering several thin quads. Stencil-based algorithms are techniques that allow write operations to only certain portions of the video memory.

4.2 Memory-Efficient Algorithm

We present an improved algorithm that has a better memory access pattern. Our improved algorithm is simple and is very similar to the original Batcher's algorithm [9]. Our algorithm operates on bitonic sequences and proceeds in the same number of stages and steps as bitonic sort. At the beginning of each stage, a step is performed to reverse one of the bitonic sequences and perform comparisons using a sorting network similar to PBSN. The remaining steps are performed with coherent memory accesses i.e., in a step j, the elements corresponding to the minimum regions are compared against the elements at a distance of 2^{j-1} in one direction. Similarly, the elements corresponding to the maximum regions are compared against the elements at a distance of 2^{j-1} in the opposite direction. Fig. 4 illustrates a simple improved sorting network on 8 data values. It can be easily proved that the sorting algorithm orders the data values [11].

Our improved algorithm performs the same number of memory accesses and comparisons as bitonic sort. However, our new algorithm has a better memory access pattern than bitonic sort and can achieve higher performance. We now describe some of the optimizations used to achieve the improved performance and the relative improvements obtained using each optimization. All of the timings were computed on a Pentium IV PC with a GeForce 6800 Ultra GPU.

4.2.1 Optimizations

We have performed four major optimizations to improve the performance of our algorithm.

- **Texture Access Pattern:** The texture access pattern of our new algorithm is better than bitonic sort. We analyze it based on the locality of accesses obtained during each step of a stage i. At the beginning of the stage, we may access elements at a distance of atmost 2^i for performing comparisons. In this step, bitonic sort accesses elements at a distance of 2^{i-1} for comparisons and performs better than our algorithm. However, the number of such steps is only $\log n$ and is quite small in comparison to the total number of steps (which is $\log^2(n)$). In each of the remaining steps j, our algorithm accesses elements at a distance of 2^{j-1} for performing comparisons whereas bitonic sort accesses elements at a distance of 2^j or 2^{j-1} based on whether the direction of access is reversed or not. The number of

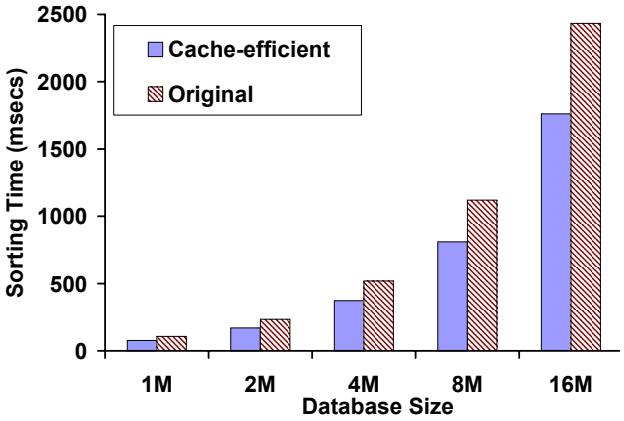

Figure 6. *Original Algorithm vs. Cache-Efficient Algorithm: This graph highlights the performance improvement obtained using our cache efficient bitonic sort algorithm. Our improved algorithm has better locality in data accesses and requires lesser number of memory accesses than the original algorithm. Our results on varying database sizes indicate a* $30-35\%$ *improvement in computational performance.*

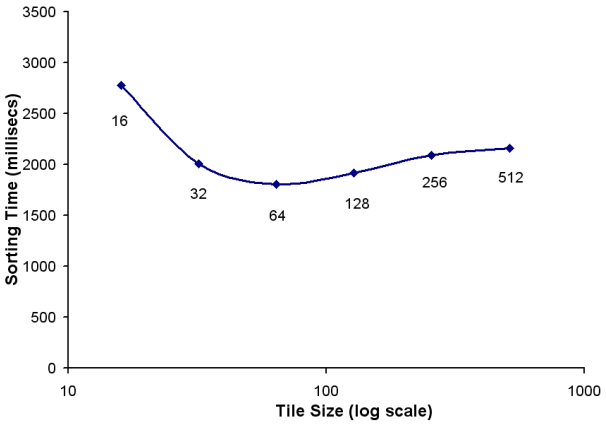

Figure 7. *Influence of tiling on sorting performance: The cache efficiency of our sorting algorithm is dependent upon the size of cache. Moreover, the cache efficiency can be improved using a tiling mechanism. This graph highlights the performance improvement obtained using tile-based algorithm on an input database of* 16 *million floating point values. We have varied the tile-sizes and measured the performance improvement obtained on the same database. We observe that at lower tile-sizes, the additional vertex rasterization overhead dominates the improvement obtained in memory accesses. At higher tile-sizes, we observe a degradation in performance due to more cache misses. The optimal performance is obtained at a tile size of* 64×64.

such steps is large and is the common case. Therefore, in practice, our algorithm incurs lower cache penalties than bitonic sort.

- **Reducing Memory Accesses:** Our GPU-based sorting algorithm is memory-limited. During each step, the sorting algorithm has the additional overhead of copying data values from the texture into a frame buffer. The copy operation is essential as blending hardware compares an input data value only against a data value stored in the frame buffer. At the end of each step, we store the frame buffer values into a separate texture. In order to perform the computation efficiently, we alternate between two render targets (or frame buffers) every step. We have further optimized our algorithm to reduce the number of memory accesses. We observe that only 50% of the data values from a texture need to be copied into the frame buffer. These data values either correspond to the regions where minimum values are stored or the maximum values are stored during a step. A simple algorithm is applied to update these regions. We have also simulated the blending functionality in the programmable pipeline, and the resulting algorithm completely eliminates the need for copying data values, thus improving the overall memory performance.

- **Tiling:** As described in Section 3.3, when a quad is drawn, it is decomposed into pixel tiles and the rasterization hardware fetches blocks of texture memory that correspond to the texels within these pixel tiles. Our computations are performed on column-aligned quads or row-aligned quads (as shown in Fig. 5). Therefore, it is more efficient to rearrange the update of these regions to account for spatial locality and improve the cache efficiency. We propose a simple tiling technique to improve the locality of accesses. When performing computations on column-aligned quads, we divide each quad into multiple quads using a constant tile height. We then perform computations on all the decomposed quads based on their row-order. The rasterization of such decomposed quads involves an additional overhead of performing vertex rasterization. Therefore, if the tile-height chosen for decomposition is too small, the overhead dominates the benefit of efficient memory accesses. On the other hand, if the decomposition tile-height is too large, there are more cache-misses. The optimal cache-size corresponds to the block sizes in the texture memory (as shown in Fig. 7).

- **Improve the Common Case:** In the each stage, as the step number decreases, the width of the quads corresponding to the minimum regions and maximum regions becomes small and finally approaches unity. In these lower steps, the vertex rasterization overhead dominates the computation time irrespective of the tile-height used for tile-based memory optimizations. In order to improve the performance in these lower steps, we apply stenciling techniques and draw only two screen-filling quads instead of several thin quads.

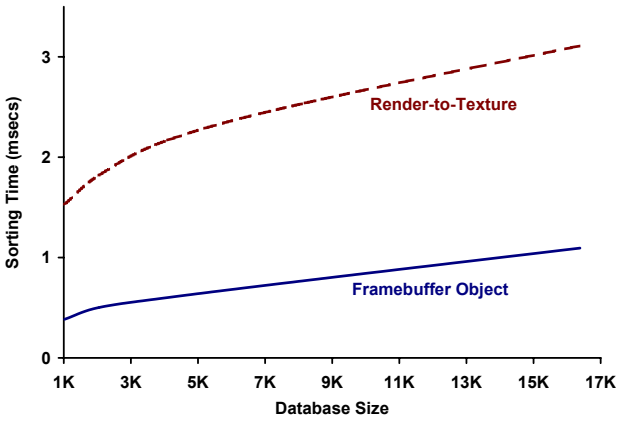

Figure 8. *Preliminary Improvements using Newer Extensions: This graph compares the performance obtained using the current render target extensions against newer extensions such as $EXT\_Framebuffer\_object$. We have used a pre-release driver 7762 from NVIDIA corporation. We have measured the sorting time on small data sizes up to 16K data values. Our preliminary results indicate a factor of 3 improvement using the newer extensions.*

We pre-compute a stencil value and store it at each location in the stencil buffer of the screen. The stencil is a mask that can be used to allow or disallow writes to certain portions of the frame buffer. For a more detailed description on stencil-based functionality, refer to [18]. We illustrate a simple example on step 1 to explain the functionality. In this step, our algorithm renders column-aligned lines. There are $\frac{W}{2}$ lines that correspond to the minimum regions and $\frac{W}{2}$ lines that correspond to the maximum regions. We store a stencil value of 1 in the regions corresponding to the maximum values and 0 in the regions corresponding to the minimum values. At run-time, we update only the minimum regions by allowing writes to the portions corresponding to a stencil value of 0. The maximum values are updated accordingly. The overhead of stenciling is negligible and due to the pixel-tile-based rendering model of graphics processors, the memory accesses are automatically localized.

Fig. 6 highlights the performance improvement obtained using our algorithm on varying database sizes. We have used an optimal tile-size of 64×64, along with the above optimizations to improve the memory efficiency and performance. Our results indicate a $30 - 35\%$ improvement in performance as compared to the original texture mapping and blending-based algorithm [19].

4.2.2 Other Factors

We have also analyzed two other factors that affect the performance of our algorithm including other GPU-based sorting algorithms.

- **Instruction Dispatch:** The number of state changes and draw commands sent to the graphics processor can affect the performance of any rendering algorithm. We have tested our algorithm using two configurations:

  1. Perform one minimum computation on a region followed by a maximum computation on a neighboring region. This configuration involves one state change per draw call and we need to perform a number of draw calls to update the minimum regions and maximum regions.
  2. Perform all minimum computations by using only one draw call. This involves only one draw call and a state change.

  Our experiments indicate that the performance improves by reducing the number of instructions dispatched to the GPU.

- **Render Target Overhead:** The current implementation of render targets on GPUs involves a significant computational overhead whenever a texture is bound to a texture unit. The number of such texture binding operations is proportional to the number of steps in the sorting algorithm and may be expensive [16]. A new specification called frame buffer objects has been recently proposed to overcome the render target implementation overhead. We have performed experiments on a pre-release driver obtained from NVIDIA Corporation. Our results (shown in Fig. 8) indicate a further performance improvement of upto a factor of 3 over our current implementation.

5 Applications and Results

In this section, we describe two common database and data mining applications whose performance can be significantly accelerated using our new GPU-based sorting algorithm. These include nested join operations and stream-based frequency computations on sliding windows. We also compare their performance against prior CPU-based and GPU-based sorting algorithms. Finally, we highlight other database and data mining algorithms that can be accelerated using our algorithm.

5.1 Sorting Analysis and Performance

In this section, we present the theoretical and practical performance of our algorithm, and compare it prior optimized sorting algorithms. We will also analyze the bandwidth requirements, and the performance growth rate of our algorithm.

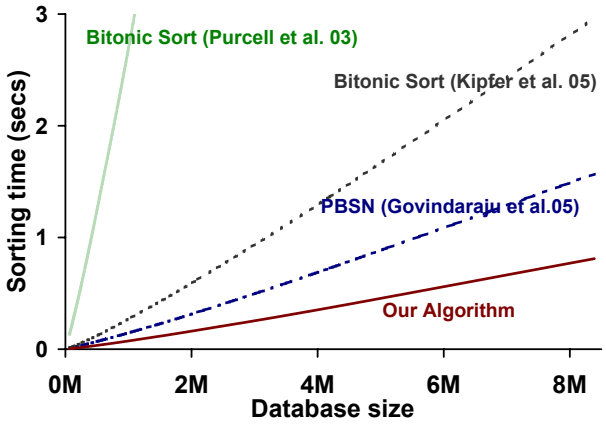

Figure 9. *Comparison with Other GPU-based Algorithms: We have compared the performance of our new algorithm with other GPU-based algorithms. Our results indicate a factor of $2-2.5$ performance improvement over GPU-based PBSN algorithm [19], 4 times improvement over improved bitonic sorting algorithm [24] and $20-30$ times performance improvement over prior GPU-based bitonic sorting algorithms [35].*

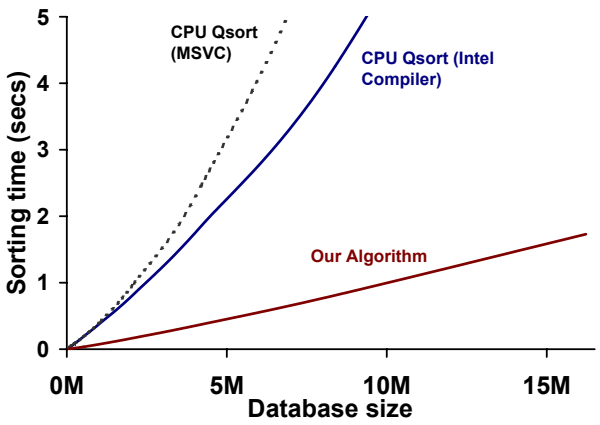

Figure 10. *Comparison with CPU-based Algorithms: This graph highlights the performance of our algorithm against optimized Quicksort implementations on CPUs. We have compared our algorithm against two implementations of Quicksort on a 3.4 GHz Pentium IV CPU. Our results indicate a $6-25$ times speedup over the Quicksort algorithm.*

5.1.1 Performance

Our algorithm has better theoretical performance than prior optimized GPU-based algorithms such as PBSN [19]. It requires lower number of memory accesses and comparison operations. Fig. 9 compares the performance of our algorithm against prior fast GPU-based algorithms. We have varied the database sizes up to 8 million values and compared the performance. Our results indicate that our new GPU-based algorithm is atleast $2-2.5$ times faster than PBSN [19], 4 times faster than the improved bitonic sorting algorithm presented in [24], and $20-30$ times faster than other bitonic sort algorithms [35]. We have also compared the performance of our algorithm against optimized CPU implementations. Fig. 10 highlights the performance improvement obtained by using our GPU-based algorithm against Quicksort. We have observed a factor of $6-25$ times speedup over optimized CPU implementations. Overall, our new algorithm is able to sort nearly 10 million floating point keys per second. In addition, our preliminary results on frame buffer objects indicate further performance improvements.

5.1.2 Bandwidth

Our algorithm has low bandwidth requirements. Given n data values, we transfer the data to and from the GPU. Our overall bandwidth requirement is O(n), and is significantly lower than the total computational cost i.e., O($nlog^2(n)$). In practice, we have observed that the data transfer time is less than 10% of the total sorting time, and indicates that the algorithm is not bandwidth limited.

5.1.3 Growth Rate

Our algorithm maps well to the graphics pipeline and the performance growth rate of our algorithm is only dependent upon the growth rate of GPUs. We have measured the growth rate of our algorithm on two high end GPUs in successive generations: NVIDIA GeForce 6800 Ultra and the NVIDIA GeForceFX 5900 Ultra. We have simulated the blending functionality using single instruction fragment programs on the NVIDIA GeForceFX 5900 Ultra GPU. Fig. 11 indicates the performance improvement of our algorithm on these GPUs for varying data sizes. We observe that the performance of our algorithm improved atleast 3 times during the past one year, and is more than the Moore's law for CPUs. Moreover, the GPU performance growth rate is expected to continue for many more years.

5.2 Applications: Join Operations

We have improved the performance of join operations by applying our fast GPU-based sorting algorithm and compared its performance against SIMD optimized nested join algorithms [40]. Zhou and Ross [40] present three different SIMD implementations:

- *Duplicate-outer*: The keys in the outer relation of the nested join are compared in parallel against the keys in the inner relation,
- *Duplicate-inner*: The keys in the inner relation are compared in parallel against each key of the outer relation, and
- *Rotate-inner*: Both inner and outer loops are SIMD parallelized and keys are compared against each other by performing rotations in the inner relation.

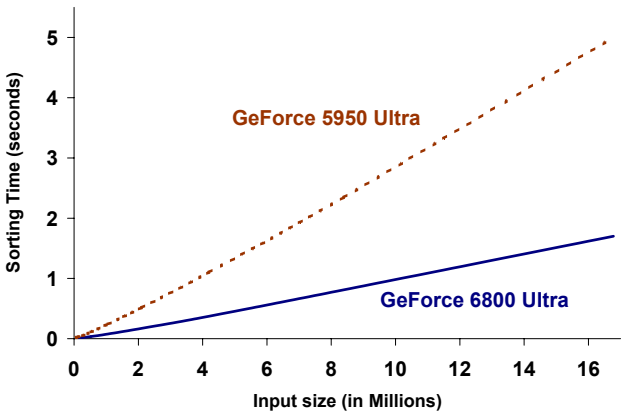

Figure 11. *Super-Moore's Law Performance: This graph highlights the performance improvement of our algorithm on two high end GPUs in successive generations. We observe a performance improvement of 3 times during the last one year.*

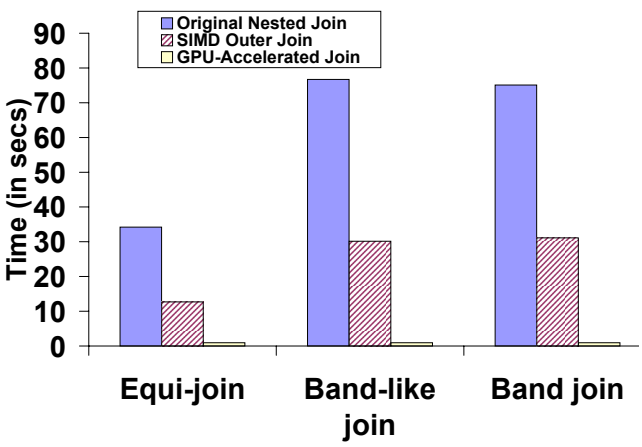

Figure 12. *Application to Join Computations: In this graph, we compare the performance of our algorithm against optimized nested join algorithms on CPUs [40]. We have used SIMD-intrinsic instructions available in Intel C++ compiler 8.0 to improve the performance of Duplicate-Outer nested join algorithm. The graph highlights the performance on an equi-join query (Query 1), a band-like-join query (Query 2) and a band-join query (Query 3).*

Experiments indicate that these SIMD-based algorithms significantly reduce the number of branch mispredictions in nested join algorithms and can result in performance improvements of up to an order of magnitude. Using our fast sorting algorithm, we present an alternate algorithm to efficiently compute equi-joins and non-equi-joins. Similar to the duplicate-outer join algorithm[40], we compute bit-vectors representing the active records of each relation. We sort both the relations based on the join keys. We use our fast sorting algorithm to order the relations. We then compute the bit-vectors efficiently using fast search algorithms such as binary search. In practice, the size of these bit-vectors is small as join-based selectivities are typically small. Therefore, the computation of bit-vectors can significantly reduce the computational overhead. We have implemented our join-based algorithm and evaluated its performance on queries proposed by Zhou and Ross [40]. In particular, we have used three queries - an equi-join query, a band-like join query and a band-join query. These queries are performed on two relational tables with the outer relation consisting of a million values and the inner relation consisting of 10^4 values. The equi-join query uses a selectivity of 10^{-4} and the band-join queries have a selectivity of 5×10^{-3}. Fig. 12 compares the performance of our GPU-accelerated algorithm against a nested-join algorithm and the duplicate-outer join algorithm. Our results indicate an order of magnitude performance improvement over the prior nested-join algorithms.

5.3 Applications: Sliding Window Algorithms

Stream statistics computations in the sliding window model has a number of applications in networking, databases, data mining, etc. These applications use limited amount of memory and are usually computation-intensive. Recently, efficient algorithms have been proposed to perform frequency and quantile computations on large data streams [7, 20]. In these algorithms, sorting is used to efficiently compute a summary data structure and used to approximately estimate the stream statistics. In practice, sorting can be a major bottleneck in these algorithms [19]. A fast GPU-based PBSN algorithm is used to accelerate the computation of summary structures, and the resulting algorithm achieves better performance over prior CPU-based algorithms.

Although more efficient than CPU-based algorithms, the PBSN algorithm has a major limitation in comparison to bitonic sort. In summary structure computations, current algorithms perform sorting of a window of data values at multiple levels as shown in Fig. 13. As the data values in the windows corresponding to the bottom levels are already sorted, we can improve the performance of sorting algorithm on the higher levels by using a fast merging routine. Unfortunately, there is no obvious modification to improve the performance of PBSN for merging sorted sequences. On the other hand, bitonic merge can sort two sorted input sequences of length n in $O(n \log n)$ whereas PBSN can require $O(n \log^2 n)$ operations. Furthermore, the bitonic sort algorithm is inplace and requires no additional memory to perform the merge operation.

Fig. 14 compares the performance of our new algorithm against GPU-based PBSN algorithm [19]. We have measured the improvement computationally and also the total time (which includes the CPU-based quicksort at lower levels and data transfer overhead). We observe a factor of $4-5$

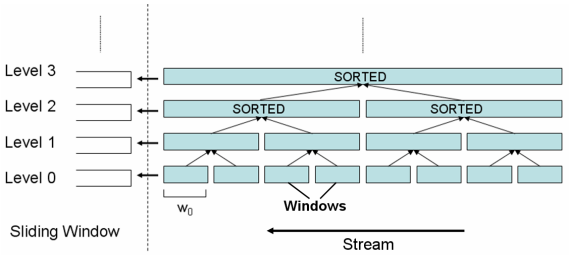

Figure 13. *Stream Statistics in Sliding Windows: A summary structure is efficiently computed at various levels as shown in the figure. Each level maintains a window size and an error factor to compute and update the summary structure. Furthermore, histogram computations are performed on each window using a sorting algorithm and each level proceeds independently [7]. However, a more efficient merge-based algorithm can be used to combine the sorted sequences at lower levels for histogram computations at higher levels. We apply our bitonic merge algorithm to efficiently compute the summary structure.*

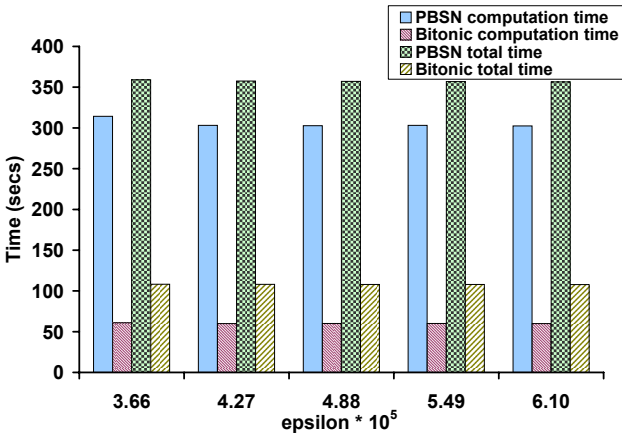

Figure 14. *Performance of bitonic sort algorithm on stream statistic computations: This graph highlights the performance of our efficient bitonic-merge-based algorithm against a prior optimized GPU-based PBSN algorithm [19]. We have measured the performance on a large incoming data stream of size $100M$ elements and used a sliding window of size $10M$. The graph highlights the performance improvement based on computation-time as well as total time. The total time includes the data transfer time, and CPU-based sorting time at the lower data levels. Overall, we observed a $4-5$ times performance improvement using our GPU-based cache-efficient sorting algorithm.*

times improvement in the performance against prior algorithms.

5.4 Other Related Applications

Many other database and data mining algorithms can be accelerated using our GPU-based cache-efficient sorting algorithm. These include hierarchical heavy hitter computation [12, 13], correlated sum computation [6], approximate join processing algorithms based on histogram computation [14], similarity-based pattern queries in continuous streams [17], etc. In addition, the performance of frequency and quantile-based algorithms [19] can also be improved. Furthermore, our cache optimizations are applicable to other applications such as fast fourier transform computations, linear systems, discrete cosine transform computations etc.

6 Analysis

In this section, we highlight three main factors that affect the performance of our algorithm:

- **Rasterization performance:** The computational performance of our algorithm mainly depends upon the number of comparison operations that can be performed per second. This is mainly governed by the cost of blending. In our theoretical analysis, we observed a $5:1$ ratio between the number of memory access and comparison operations. Therefore, based on the memory access time and the compute-time on different GPUs, the algorithm can be either memory-limited or compute-limited.

- **Memory-based tile sizes:** The tile sizes used for block-fetches within the texturing hardware can influence the sorting performance. Also, based on texture pre-fetching schemes, the performance can be further improved. Moreover, efficient memory representations such as compressed frame buffers can improve the performance.

- **Pixel-tile sizes:** The performance at the lower steps of the algorithm depends on the size of the pixel tiles. These lower steps occur frequently and usually result in a large fraction of partially covered tiles, which can be computationally inefficient. It may be possible to use stenciling techniques on the lower steps to improve the performance.

7 Conclusions and Future Work

We have presented a cache-efficient sorting algorithm that maps well to the GPUs. We analyzed the memory-limitations of current GPU-based sorting algorithms and presented techniques to improve the computational performance. Furthermore, our new sorting algorithm performs fewer number of memory accesses and exhibits better locality in data access patterns. It takes into account the sorted nature of the input sequence and improves the overall performance. We have demonstrated its application on some database and data mining applications. Our results indicate a significant improvement over prior techniques.

There are many avenues for future work. We would like

to investigate new features and capabilities of GPUs that can further improve the performance, and compare against other optimized algorithms such as hash-joins on CPUs. We would like to use our sorting algorithm for other data mining applications. Furthermore, we would like to develop cache-friendly GPU-based algorithms for other computations including nearest neighbors and support vector machine computations.

References

[1] Ramesh C. Agarwal. A super scalar sort algorithm for RISC processors. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):240–246, June 1996.

[2] A. Ailamaki. Database architectures for new hardware. *VLDB Tutorial Notes*, 2004.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases*, pages 169–180, 2001.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *The VLDB Journal*, pages 266–277, 1999.

[5] Tomas Akenine-Möller and Jacob Strom. Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Trans. Graph.*, 22(3):801–808, 2003.

[6] R. AnanthaKrishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Shrivastava. Efficient approximation of correlated sums on data streams. *TKDE*, 2003.

[7] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. *PODS*, 2004.

[8] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. *VLDB*, 2004.

[9] K.E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, 1968.

[10] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, pages 54–65, 1999.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[12] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB 2003: Proceedings of 29th International Conference on Very Large Data Bases*, pages 464–475, 2003.

[13] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Diamond in the rough: finding Hierarchical Heavy Hitters in multi-dimensional data. In ACM, editor, *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data 2004, Paris, France, June 13–18, 2004*, pages 155–166, New York, NY 10036, USA, 2004. ACM Press.

[14] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 40–51. ACM Press, 2003.

[15] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. *Proc. of Graphics Hardware*, 2004.

[16] EXT_Framebuffer_Object OpenGL Specification. http://www.opengl.org/documentation/extensions/EXT_framebuffer_object.txt.

[17] Like Gao and X. Sean Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 370–381. ACM Press, 2002.

[18] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. *Proc. of ACM SIGMOD*, 2004.

[19] N. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *Proc. of ACM SIGMOD*, 2005. to appear.

[20] M. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. *PODS*, 2004.

[21] Z. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. *Proc. of 24th International Symposium on Computer Architecture*, pages 108–120, 1997.

[22] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 133–ff. ACM Press, 1998.

[23] B. Kelleher. Pixelvision architecture. Technical Report 1998-013, Digital Systems Research Center, 1998.

[24] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A gpu-based particle engine. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2004.

[25] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[26] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. *Proc. of SODA*, pages 370–379, 1997.

[27] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 339–350, 2000.

[28] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB 2002: proceedings of the Twenty-Eighth International Conference on Very Large Data Bases*, pages 191–202, 2002.

[29] D. Manocha. *Interactive Geometric and Scientific Computations using Graphics Hardware*. SIGGRAPH Course Notes # 11, 2003.

[30] Joel McCormack and Robert McNamara. Tiled polygon traversal using half-plane edge functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 15–21. ACM Press, 2000.

[31] Shintaro Meki and Yahiko Kambayashi. Acceleration of relational database operations on vector processors. *Systems and Computers in Japan*, 31(8):79–88, August 2000.

[32] S. Morein. ATI Radeon HyperZ technology. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Hot3D Proceedings*, 2000.

[33] NV40- GeForce 6800 ultra review. http://www.beyond3d.com/previews/nvidia/nv40/.

[34] NVIDIA GeForce 6800 Ultra - part one architecture features. http://www.digit-life.com/articles2/gffx/nv40-part1-a.html.

[35] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 41–50, 2003.

[36] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, pages 78–89, 1999.

[37] Kenneth A. Ross. Conjunctive selection conditions in main memory. In ACM, editor, *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2002: Madison, Wisconsin, June 3–5, 2002*, pages 109–120, New York, NY 10036, USA, 2002. ACM Press. ACM order number 475021.

[38] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 510–521, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

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

[40] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In Michael Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pages 145–156, New York, NY 10036, USA, 2002. ACM Press. ACM order number 475020.

[41] Jingren Zhou and Kenneth A. Ross. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 191–202. ACM Press, 2004.