Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors

Naga K. Govindaraju Nikunj Raghuvanshi Dinesh Manocha
University of North Carolina at Chapel Hill
{naga,nikunj,dm}@cs.unc.edu
http://www.cs.unc.edu/gamma/STREAMING

ABSTRACT

We present algorithms for fast quantile and frequency estimation in large data streams using graphics processors (GPUs). We exploit the high computation power and memory bandwidth of graphics processors and present a new sorting algorithm that performs rasterization operations on the GPUs. We use sorting as the main computational component for histogram approximation and construction of ϵ-approximate quantile and frequency summaries. Our algorithms for numerical statistics computation on data streams are deterministic, applicable to fixed or variable-sized sliding windows and use a limited memory footprint. We use GPU as a co-processor and minimize the data transmission between the CPU and GPU by taking into account the low bus bandwidth. We implemented our algorithms on a PC with a NVIDIA GeForce FX 6800 Ultra GPU and a 3.4 GHz Pentium IV CPU and applied them to large data streams consisting of more than 100 million values. We also compared the performance of our GPU-based algorithms with optimized implementations of prior CPU-based algorithms. Overall, our results demonstrate that the graphics processors available on a commodity computer system are efficient stream-processor and useful co-processors for mining data streams.

Keywords: data streams, graphics processors, quantiles, frequencies, sorting, memory bandwidth, sliding windows

1. INTRODUCTION

Many real-world applications such as high-speed networking, finance logs, sensor networks, and web tracking generate massive volumes of data. This data is collected from different sources and is modeled as an unbounded data sequence arriving at a port on a system. Typically, the size of a *data stream* is so large that it may not be possible to store the entire stream in main memory for online processing. Due to the data stream's continuous nature, the underlying application performs continuous queries on the data stream as opposed to traditional one-timed queries.

The problem of computing over data streams has received interest in many areas including databases, computer networking, computational geometry and theory of algorithms [41]. Some of the fundamental mathematical problems in data streaming include computing the number of distinct items, quantiles and frequencies [6, 23, 26, 28, 34, 35, 40]. The quantile and frequency estimation algorithms have also been used as subroutines to solve more complex problems related to histogram maintenance and dynamic geometric computations [25].

In data streaming applications, each data element is accessed at its arriving time and then needs to be processed in real time. Furthermore, a data processing algorithm needs to use the small size memory footprint as possible. It can be challenging to satisfy these constraints, especially when there are irregularities and bursts in data arrival rates. As a result, the underlying data stream management system (DSMS) [5, 41] can become resource limited. This problem arises due to insufficient time for the underlying CPU to process each stream element or insufficient memory to process the queries. In such cases, some DSMSs resort to load-shedding [5], i.e. dropping excess data items. The other option is to allow spilling of data items to the disks and use appropriate memory hierarchies to speed up the overall system performance [39]. Ideally, we would like to develop new hardware-accelerated solutions that can offer improved processing power and memory bandwidth to keep up with the update rate.

1.1 GPUs for Data Streaming

In this paper, we exploit the inherent parallelism and high memory bandwidth of graphics processing units (GPUs) for quantile and frequency computations on data streams. GPUs have been traditionally designed for fast rendering of geometric primitives for visual simulation and computer gaming; they are now a part of every PC, game console, and laptop. Recently, GPUs are also being designed for handheld devices and mobile phones.

Modern GPUs feature programmable vertex and fragment processors. These GPUs can be thought of as a particular kind of stream processors: operating on data streams consisting of an ordered sequence of attributed primitives including vertices or fragments. As compared to conventional CPUs, GPUs consist of high-bandwidth memories and more floating-point hardware units. For example, a current top of the line GPU, NVIDIA 6800 Ultra, has a peak performance of 45 GFLOPS and memory bandwidth of 36 GB/sec, as compared to 12 GFLOPS and 6 GB/sec, respectively, for a 3 GHz, Pentium IV CPU. Furthermore, the GPUs performance has been growing at a rate of $2.5-3.0$ times a year, which is faster than the Moore's Law for CPUs [37]. Given the programmability and computational capabilities of GPUs, many GPU-based algorithms have been designed for scientific and geometric computations, global illumination, and database operations [37, 32].

1.2 Main Results

We present algorithms for the fast processing of quantiles and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2005, June 14-16, 2005, Baltimore, Maryland, USA
Copyright 2005 ACM 1-59593-060-4/05/06 ...$5.00.

frequencies in large data streams using GPUs. Our approach is based on recent algorithms for quantile estimation [22] and frequency estimation [34] and is also applicable to hierarchical heavy hitter and correlated sum aggregate queries. We present a novel algorithm for sorting on GPUs. Our sorting algorithm is based on periodic balanced sorting network and utilizes the high computation power and memory bandwidth of GPUs. The underlying operations of comparisons and comparator mapping are performed using the color blending and texture mapping capabilities of GPUs. We use sorting as the main computational component for histogram approximation and construction of ϵ-approximate quantile and frequency summaries. Our overall algorithms for numerical statistics on data streams are deterministic and applicable to fixed or variable-sized sliding windows. We use a limited memory footprint and the overall space complexity is comparable to prior CPU-based algorithms.

We use the GPU as a co-processor for sorting computations and histogram approximation. We take into account low bus bandwidth between the CPU and GPU and minimize the data transmission between them. We have implemented our algorithms on a PC with 3.4 GHz Pentium IV processor with NVIDIA GeForce 6800 FX Ultra GPU and applied them to data streams consisting of more than 100 million values. In practice, our novel GPU-based sorting algorithm is nearly one order of magnitude faster as compared to prior GPU-based sorting algorithms. Moreover, the performance of our algorithm is comparable to one of the fastest implementations of Quicksort on a Pentium IV CPU. We also highlight the performance of our algorithm on frequency and quantile estimation over fixed and sliding windows.

Overall, we show that our GPU-based algorithms offer an excellent alternative for quantile and frequency estimation in data streams. Our algorithms are able to exploit the memory bandwidth and inherent parallelism of GPUs for fast computation. As a result, the graphics processors available on commodity computer systems are effective co-processors for mining data streams.

1.3 Organization

The rest of this paper is organized in the following manner. We give a brief overview of prior work on numerical statistics on data streams, GPU-based computations and sorting in Section 2. Section 3 gives a brief overview of quantile and frequency estimation, performance bottlenecks in CPU-based algorithms and some architectural features of GPUs. We present our novel GPU-based sorting algorithm in Section 4 and highlight some of its features. We describe our implementation in Section 5 and compare the performance of our algorithms with prior CPU-based algorithms. We analyze the performance of our algorithms and highlight some of their limitations in Section 6.

2. RELATED WORK

We give a brief summary of related work in computing numerical statistics on data streams, and the use of GPUs for general purpose computation and sorting.

2.1 Quantiles and Frequencies

Most of the research in streaming algorithms has focused on computing numerical statistics like median, quantiles, number of distinct elements, and frequencies. In this section, we give a brief overview of algorithms for approximate quantile and frequency computations on data streams.

Many algorithms have been proposed for quantile estimation in data streams. Given a sequence of N elements, an ϵ-approximate algorithm answers quantile queries about the sequence to within a precision of ϵ N. These can be classified into deterministic algorithms [23, 33, 35] and probabilistic algorithms [11, 36]. Some of the recent work has been on improving the space requirements of the deterministic algorithms [6, 46].

The problem of finding frequent items over data streams has been an active area of research in data streaming. At a broad level, different algorithms can be classified into sample-based approaches and hash-based approaches. The sample-based approaches keep track of counters [14, 28, 34]. One of the earliest sample-based deterministic algorithms for approximate frequency counts was presented by Misra and Gries [38]. Recently, Demaine et al. [14] and Karp et al. [28] re-discovered the same algorithm and reduced its worst case processing time to $O(1)$. Manku and Motwani [34] presented a new deterministic and one-pass algorithm that uses out-of-core summary structure. Jin and Agrawal [27] improved the memory requirements and described an in-core algorithm .

The hash-based approaches for frequency counts [10, 12, 18, 26] use a hash table and each item in the stream owns a respective list of counters the table. These algorithms can also handle delete operations.

2.2 Streaming Computations on the GPUs

Many researchers have advocated the use of GPUs as a stream processor [9, 17, 37, 45] for compute-intensive applications. Although GPUs are primarily designed for real-time rendering, the programmable vertex and fragment processors in the GPUs feature instruction sets are general enough. In particular, GPUs have been used for scientific computations including matrix multiplication, sparse linear systems, FFT computation, fluid dynamics and their applications to physical simulation [37], geometric computations and optimization [1, 37, 19], global illumination [42], etc.

Database Computations on GPUs: Recently, there has been interest in using GPUs to speedup database computations [7, 20, 43]. Sun et al. [43] used the rendering and search capabilities of GPUs for spatial selection and join operations on real world datasets. Bandi et al [7] integrated GPU-based algorithms for spatial database operations into Oracle 9I DBMS. Furthermore, they demonstrated significant improvement in the performance of spatial operations. Govindaraju et al. [20] presented novel algorithms for predicates, boolean combinations and aggregates on commodity GPUs and obtained considerable speedups over CPU-based implementations. These algorithms take into account some of the limitations of the current programming model of the GPUs and were applied to perform multi-attribute comparisons, semi-linear queries, range queries and kth largest numbers. Muthukrishnan [41] has advocated the use of GPUs for data streaming algorithms, though we are not aware of any GPU-based algorithms for numerical statistics on data streams.

2.3 Sorting

Sorting is a key operation used by the current algorithms for quantile and frequency computations. Sorting is a well studied problem in the theory of algorithms [30], and optimized implementations of some algorithms such as Quicksort are widely available. Many fast algorithms have also been designed for transaction processing and disk to disk sorting in the database literature [2]. However, the performance of sorting algorithms on conventional CPUs is governed by cache misses [31] and instruction dependencies [47].

In terms of using GPUs for sorting, Purcell et al. [42] described an implementation of bitonic merge sort on the GPUs. The algorithm is implemented as a fragment program and each stage of the sorting algorithm is performed as one rendering pass. Kipfer et al. [29] 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. However, these algorithms do not make full use of the rasterization capabilities and computational power of the current GPUs. Govindaraju et al. [19] have described an algorithm to sort 3D geometric primitives using GPUs.

3. BACKGROUND AND OVERVIEW

In this section, we give an overview of numerical statistics computation on data streams. We identify the main bottlenecks in CPU-based implementations of these algorithms. We also give an overview of the architectural features of current GPUs.

3.1 Terminology

A data stream is a continuous sequence of data values that arrive in time. Our goal is to design fast and deterministic algorithms to estimate quantiles and frequencies over a large data stream. Formally, we define approximate quantiles and frequencies as follows:

1. **Quantiles:** A quantile defines the position or the rank of an element in a sorted sequence of incoming elements. In a stream with N values, a ϕ-quantile is defined as an element with rank $\lceil \phi N \rceil$ and an ϵ-approximate quantile is any element whose rank is between $\lceil (\phi - \epsilon) N \rceil$ and $\lceil (\phi + \epsilon) N \rceil$ [6].

2. **Frequencies:** The frequency of an element denotes its number of occurrences in a stream. The estimated frequency $\tilde{f}$ of an element is ϵ-approximate if $\tilde{f}$ is smaller than the true frequency f of the element by at most ϵN, i.e. $\tilde{f} \geq (f - \epsilon N)$.

In this paper, we mainly focus on the following queries that arise in data streaming:

- **Quantile-Based:** Given a stream of length N and a support s, where $s \in [0, 1]$, compute the ϵ-approximate s-quantile using a limited memory footprint.

- **Frequency-Based:** Given a stream of length N and a support s, compute the ϵ-approximate frequencies of all elements whose exact frequency is above a threshold sN using a limited memory footprint.

We refer to these queries as *ϵ-approximate queries*. These queries are performed on the data stream in two possible manners:

1. **Entire past history:** Apply the query over all the elements in the past history of the stream [12, 22, 23, 27, 28, 34].

2. **Sliding windows:** Apply the query on a sliding window of recent data from the streams [5, 6, 13, 46]. These windows could be fixed or variable-sized width.

Most of the algorithms used to perform these queries use an ϵ-approximate summary data structure for efficient computation. In this paper, we present novel GPU-based features to accelerate these queries over data streams.

3.2 ϵ-Approximate Summary Computation

In this section, we give a brief overview of the basic operations involved in the construction of an ϵ-approximate summary data structure. We also analyze their performance and identify some of the bottlenecks.

The underlying algorithms use a limited memory footprint and are used to perform an ϵ-approximate query. The summary data structure is usually maintained as a sorted sequence of tuples [6, 22, 28, 34]. Each tuple has a fixed size and holds the value of an element within the stream. The tuple may also consist of additional fields such as the frequency of the element or the minimum and the maximum rank of the element. The sorted order is computed based on the element values. Sorting improves the performance as new elements from the stream can be inserted efficiently by performing a binary search.

Many other algorithms for mining data streams also compute an approximation of the histogram and use the histograms to compute the numerical statistics. Histograms are widely used to track the distribution of the data in a database and they have been extensively studied in the database literature [24]. More recently, algorithms have been proposed to compute and maintain dynamic histogram structures in a continuous data stream [44]. Given a multi-attribute data and a limited memory footprint, these algorithms construct the "best" histogram that minimizes some error metric.

The insertion of elements into the summary data structure is performed in one of following ways:

- **Single element-based:** An element is inserted into the summary structure as it arrives [13, 14, 23, 27, 28].

- **Window-based:** A subset of the elements of a window are computed and inserted into the summary structure [6, 22, 34].

The window-based algorithms usually perform better in practice as fewer number of elements are inserted into the summary data structure. Moreover, the performance of window-based algorithms can be further improved by maintaining these subsets of elements in a sorted order. However, window-based algorithms may have a slightly higher memory requirement as compared to single element-based algorithms. In the rest of this paper, we mainly focus on improving the performance of recent window-based deterministic algorithms for frequency estimation [34] and quantile estimation [22]. These algorithms for frequency and quantile estimation are also used for other numerical statistics computations over data streams [6, 12].

At a broad level, the window-based algorithms compute the ϵ-approximate summary by performing three main operations: histogram computation, merge operation and compress operation.

1. **Histogram computation:** For each window, the elements are ordered by sorting them and a histogram is computed. A histogram data structure holds each element value in the window and its frequency [44]. A subset of the histogram elements are computed and used for subsequent operations. The frequency computation algorithms use the entire histogram along with the frequencies of the elements [12, 34] for subsequent operations. On the other hand, the quantile computation algorithms compute a subset of histogram elements by sampling the sorted sequence at the rate of at least ϵW for a window of size W, and maintain the minimum and maximum ranks of the elements.

2. **Merge operation:** The merge operation inserts a subset of elements of the window into the ϵ-approximate summary data structure. The performance of merge operation is dependent upon the number of memory allocations performed, and can be quite significant for naive implementations. Many techniques are known for improving the performance of merge operations.

3. **Compress operation:** The compress operation deletes few of the elements in the ϵ-approximate summary data structure and reduces its size. Given an ϵ, the compress operation

is used in frequency and quantile computations to provide a worst-case upper bound on the memory requirements.

Among these three operations, the sorting operation used for histogram computation is the most expensive operation. In our implementation of numerical statistic algorithms, sorting can take $70 - 95\%$ of the total time with a 3.4 GHz Pentium IV processor. The performance of sorting algorithms on conventional CPUs is governed by the following reasons:

- **Cache misses:** The performance of a sorting algorithm is largely dependent on the cache efficiency, since main memory accesses can be slow. For example, the cache access times for $L1$ and $L2$ caches, and main memory are of the order of 1-2 clock cyles, 10 clock cycles and 100 clock cycles, respectively. Therefore, accesses to the main memory due to incoherent data reads and writes can often lead to a loss in performance. This is indicated by an analytical and empirical study conducted by LaMarca and Ladner [31]. They observed that the quicksort algorithm incurs one cache miss per block when the input sequence fits within the cache. For larger sequences quicksort incurs a substantially higher number of misses. Moreover, cache sizes are typically small on PCs (in the order of a few hundreds of kilobytes). For example, the 3.4 GHz Intel Pentium IV processor has a $L1$ cache of size 128 KB and a $L2$ cache size of 1 MB.

- **Instruction dependencies:** Due to the presence of conditional branches in sorting algorithms, branch mis-predictions can lead to stalls and can be quite expensive on current CPUs. For example, a Pentium IV processor has a minimum penalty of 17 clock cycles per branch mis-prediction and the stall can significantly lower the performance of common database operations [47].

3.3 GPUs and Data Streaming

In this section, we give an overview of the architectural features of GPUs. In particular, we highlight some capabilities of GPUs that are very useful for sorting.

GPUs are well-optimized to render a stream of geometric primitives onto a rectangular array of pixels. Each primitive is rasterized to generate fragments for the pixels covered by the primitive and each fragment is associated with a color, texture co-ordinates, and a depth value [15]. An array of SIMD fragment processors perform user-specified instructions and tests on these fragments, and write the results to the pixels in a frame buffer or a texture. The frame buffer and the textures are used to store the 2D color values in the video memory.

In order to use GPUs for general purpose applications, the underlying computations are performed by rendering full-screen quadrilaterals (quads). In these applications, the throughput of a GPU depends on two main factors:

1. **Parallelism:** The computation time depends on the number of fragment processing units used to perform the computation. The computing time is also a function of the number of parallel vector operations performed using the vector processing units of each fragment processor. Current GPUs consist of a large number of fragment processors, each with four vector processors. For example, the NVIDIA GeForce FX 6800 Ultra can perform 64 operations in one GPU computational clock cycle and yields a peak performance of 45 GFLOPs per second.

2. **Memory access time:** A lack of balance between the computation time and the memory access time on any processor leads to stalls in the computation. In order to avoid these stalls, the memory

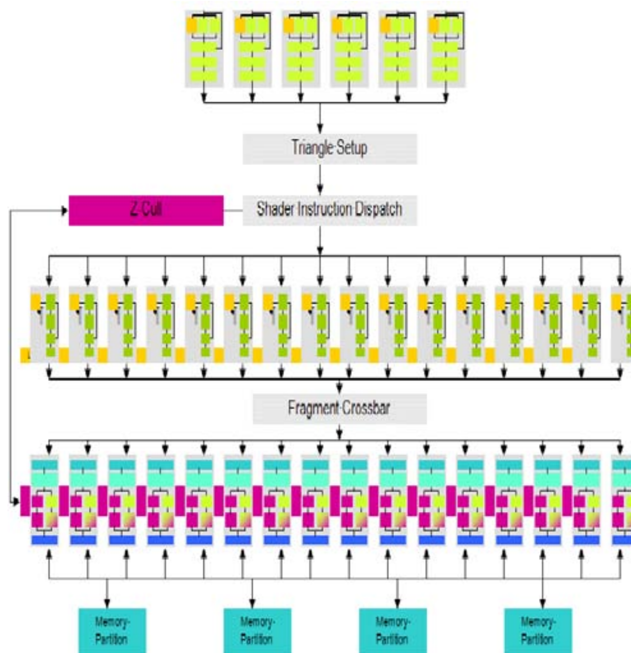

Figure 1: ***Architecture of a commodity GPU: NVIDIA GeForce 6800 FX: It has six programmable vertex processors and*** 16 ***programmable fragment processors. Furthermore, the fragment processors have a high memory bandwidth interface to the video memory. We exploit the parallelism and memory bandwidth for sorting and computing numerical statistics on data streams.***

clock of a GPU is typically designed to be many times faster than the computational clock (or core clock) of the GPU. For e.g., an NVIDIA GeForce FX 6800 GPU has a core clock of 400 MHz and a memory clock of 1.2 GHz. In contrast, the main memory clocks on a PC are typically slower than the CPU processor speeds, leading to a high penalty for cache misses on the CPUs. As an example, a high-end Pentium IV has a clock speed of 3.4 GHz and current high-end double data rate (DDR) II main memories have memory speeds of 533 MHz. Moreover, the memory interface between the GPU and the video memory on the graphics card is rather wide (in terms of number of bits). This allows each pixel processor to access the data from the video memory in parallel. For example, NVIDIA GeForce FX 6800 Ultra has a 256 bit memory interface to its video memory and can access 88 bytes of data in one GPU computational clock cycle, thus effectively yielding a peak memory bandwidth of 35.2 GB per second. On the other hand, current PCs support DDR II main memories with a relatively much lower peak bandwidth of 6 GB per second. A conceptual representation of the video memory on this GPU is shown in Fig. 1. Conceptually, the video memory in the GPU is similar to a significantly large $L1$ or $L2$ cache in terms of memory bandwidth and with some restrictions on data access.

4. SORTING ON GPUS

In this section, we present a novel GPU-based sorting algorithm. We exploit the computational power and high memory bandwidth of GPUs to sort a stream of data values quickly. The GPUs are well-optimized for rasterization and allow most rendering applications to achieve close to the peak memory performance. In order to obtain high computational performance on GPUs, we use a sorting network based algorithm and each stage is performed using rasterization. In the rest of this section, we give an overview of our

data representation and the underlying GPU operations. After that we present our sorting algorithm and compare its performance with prior sorting algorithms.

Most sorting algorithms require the ability to write to arbitrary locations. The GPUs do not allow a fragment processor to write at any arbitrary memory location, i.e. the "scatter" operation is not supported. The main reason for this limitation is that GPUs avoid possible *write-after-a-read* hazard between multiple fragment processors that may be accessing the same memory location. At the same time, this limitation makes it hard to implement popular sorting algorithms such as Quicksort on GPUs.

4.1 Data Representation and Transmission

We first provide a brief overview of the data representation used by our algorithm on GPUs. Data is stored on a GPU in the form of textures. A texture is a 2D array of data values and contains multiple channels. Current GPUs support four channels in each texture value: red, green, blue and alpha (RGBA). A texture of width W and height H can store $W * H$ color values in each channel of the texture. These data values can be represented as integers or floating point values with 32 bit precision.

Data is usually sent from the CPU to the GPU in the form of textures. The transfer is performed over a data bus that connects the GPU to the CPU and the communication cost is dependent upon bus bandwidth. On current PCs, an AGP 8X or a PCI-E bus is used to perform the communication. The data bus can achieve a theoretical peak bandwidth rate of $2 - 4$ GBps. In practice, the data transfer rates are much lower ($\sim 800MBps$) and there can be a significant overhead in the computation time due to the slow data transfer. Our goal is to utilize the high memory bandwidth of GPUs to perform computations on the data streams. In order to overcome these limitations, we stream the data once to the GPU, perform the computation, and readback the data back to the CPU and avoid multiple data transfers whenever possible. Given a window of elements in a data stream, we pack the data values into a 2D texture and transfer it to the GPU. Current GPUs can represent data values in 32-bit IEEE single precision floating point representation. In order to utilize the parallelism offered by the four vector processing units in each fragment processor, we buffer four windows of data values and represent each of the windows in a color component of the 2D texture. Each window of data value is sorted in parallel and we merge the four sorted lists back on the CPU. The 2D texture is initially transferred to the GPU, and copied into a frame buffer. Sorting operations are performed on the four color components of the texture simultaneously, and the sorted data is stored in the frame buffer or a rendered texture. Finally, the sorted texture is readback to the CPU.

4.2 Texture Mapping and Blending

Our algorithms for sorting and numerical statistics make use of two main features of GPUs. These are texture mapping hardware and color blending. We perform comparison operations and comparator mapping using these capabilities.

4.2.1 Texture Mapping

Graphics processors have specialized texture mapping hardware that is designed to assign the color of the fragments of a primitive. The fragment color is assigned by performing a 2D look-up based on the fragment's texture co-ordinate in a 2D image (known as a texture). We present a brief overview of these computations and illustrate them with a simple example. Each vertex of the 2D primitive/quad is assigned a texture co-ordinate, and the texture co-ordinate of a fragment of the quad is computed using bi-linear interpolation of the texture co-ordinates of the vertices.

Routine 4.1 shows a simple example to copy a list of numbers into a frame buffer. Given a list of numbers of size n, the numbers are represented in a 2D array of width $W = \lfloor 2^{\frac{\log n}{2}} \rfloor$ and height $H = \lceil 2^{\frac{\log n}{2}} \rceil$ and stored as a 2D texture on the GPU. To copy the numbers into the frame buffer, we enable texturing on the GPU and set the 2D texture as the current active texture. Next we draw a quad with the vertices and the texture co-ordinates set to the corners of the quad i.e., each vertex is set to the same position and texture co-ordinates and correspond to the values $(0, 0)$, $(W, 0)$, (W, H) and $(H, 0)$. The color values in the frame buffer now correspond to the values in the texture.

```
Copy( tex, W, H)
1  Enable Texturing and set tex as active texture
/* Set the Texture Co-ordinates t[4], Vertex Co-ordinates v[4]*/
2   v[0] = (0,0), t[0]= (0,0)
3   v[1] = (W,0), t[1]= (W,0)
4   v[2] = (W,H), t[2]= (W,H)
5   v[3] = (0,H), t[3]= (0,H)
6  DrawQuad(v,t)
```

ROUTINE 4.1: *The routine* `Copy` *is used to copy a set of input values stored in a 2D texture* `tex` *into the frame buffer.*

The performance of texture mapping is enhanced on GPUs by using fast texture caches to save the memory bandwidth.

4.2.2 Blending

Blending operations are a set of user-specified instructions in the fragment processors that are designed to manipulate the color component of the fragments. When we enable blending, each fragment processor executes blending instructions in parallel on different fragments. The input to a blending operation is the fragment color and the corresponding color values of the pixel that is stored in the frame buffer. GPUs support a wide variety of blending operations, including conditional assignments to the fragment color. The conditional assignment is a *vector* operation and can perform comparisons between the four color components (i.e. RGBA) of the two inputs at each fragment simultaneously. The conditional assignment stores either the minimum or the maximum of these color components in the frame buffer or a texture. Each conditional assignment operation is highly optimized for performing these vector operations on GPUs.

Routine 4.2 shows an example of using a conditional assignment on a list of numbers. Given an array of n numbers (n is even), we compute the minimum of the i-th number and $(n - i)$-th number, $i < \frac{n}{2}$, and store the minimum in the location of the i-th number. On a GPU, we represent these numbers as a portion of the 2D texture with texture co-ordinates $(0, s)$, (W, s), $(W, s + H)$, $(0, s + H)$. First, a copy operation is performed to copy the texture into the frame buffer. To perform the conditional assignment, we enable blending and set the blending function to store the minimum. We draw a quad filling the first half of the 2D texture, and the texture co-ordinates are set to the reverse of the second half of the texture. That is, each vertex of the quad is assigned the position $(0, s)$, (W, s), $(W, s+\frac{H}{2})$, $(0, s+\frac{H}{2})$ and the texture co-ordinates are set to $(W, s + H)$, $(0, s + H)$, $(0, s + \frac{H}{2})$, $(W, s + \frac{H}{2})$. The minimum color values are stored in the appropriate locations in the frame buffer. We use this routine within our GPU-based sorting algorithm.

4.3 Sorting Networks on GPUs

We use techniques based on *sorting networks*. Sorting networks are a class of sorting algorithms that map well to mesh-based architectures [8, 16]. A sorting network is comprised of a set of two-input and two-output comparators. They take as input an unordered

```
ComputeMin( tex,s, W, H)
1  Enable Texturing and set tex as active texture
2  Enable Blending and set blend function to compute the minimum
/* Set the Texture Co-ordinates t[4], Vertex Co-ordinates v[4]*/
3  v[0] = (0, s),          t[0]= (W, s+H)
4  v[1] = (W, s),          t[1]= (0, s+H)
5  v[2] = (W, s + H/2), t[2]= (0, s+H/2)
6  v[3] = (0, s + H/2),  t[3]= (W, s+H/2)
7  DrawQuad(v, t)
```

ROUTINE 4.2: *The routine* `ComputeMin` *is used to compute the minimum of the value at the i-th location,* $i < \frac{W*H}{2}$ *and the value at the* $(W * H - i)$*-th location of a texture. Initially, the texture values are copied into the frame buffer using the routine* `Copy`*. The texture is set as the active texture (line 1), and the blend function is set to compute the minimum (line 2). A quad with half the height of the texture is drawn with appropriate texture co-ordinates (lines 3 - 7) and the minimum is stored at the* i*-th location of the frame buffer.*

set of items on the input ports. The output on the output ports corresponds to the smallest value on the first port and the second smallest value on the second port, etc. The sorting time, i.e. the time needed for the values to appear at the output ports, is a function of the number of stages of the network. A stage is a set of comparators that are all active at the same time. The unordered set of items are input to the first stage; the output of the ith stage becomes the input to the $(i + 1)$th stage.

In a mesh-based architecture, processors are laid out on a rectangular mesh. The layout of the pixels on the screen is similar to a rectangular mesh and hence, algorithms based on sorting networks map well to GPUs. Sorting network algorithms proceed in multiple stages and during each stage, a comparator mapping is created in which every pixel on the screen is compared against exactly one other pixel on the screen. For each pair of the mapped pixels, a deterministic order for storing the output value is defined. The minimum is stored in one of the two pixels and the maximum is stored in the other.

In order to implement these algorithms on GPUs, two fundamental operations are required:

- **Comparisons:** A basic requirement for most sorting algorithms is the ability to compare values. On graphics processors, we use the blending operations to perform comparisons efficiently. More specifically, we perform conditional assignments on the pixels and store either the minimum or the maximum for each comparison in the sorting network.

- **Mapping:** In each stage of a sorting network, a comparator mapping is used. On GPUs, the comparator mapping is performed using the texture mapping functionality of the GPU.

Using these two functionalities, we can design optimal sorting networks such as bitonic sort [8] and periodic balanced sort [16] efficiently using any traditional GPU. In each stage, the comparators in the sorting network are defined using appropriate texture co-ordinates. For each pixel, we use the texture mapping functionality to obtain the fragment color that needs to be compared against. We use the blending operation to compare the pixel color against the fragment color. We update the portions of the screen where the minimum values are to be stored by setting the blend function to output the minimum. Then quads with appropriately assigned texture co-ordinates are drawn on these portions of the screen. We perform the maximum computation in a similar manner.

4.4 Periodic Balanced Sorting Network

Our sorting algorithm is based on the periodic balanced sorting network (PBSN) [16]. We sort the input in at most $O(log^2 n)$ steps.

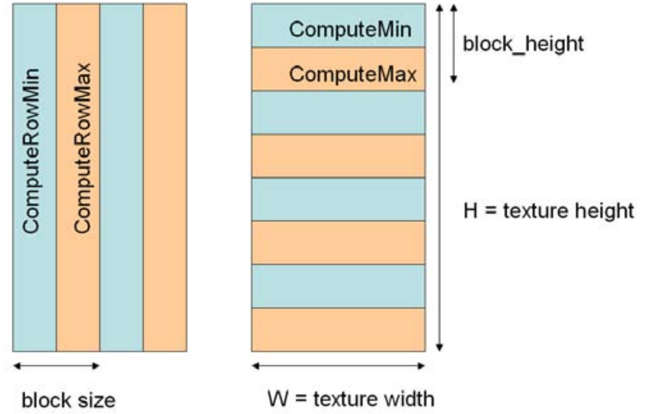

Figure 2: ***Sortstep computation in PBSN sorting algorithm: We use two cases based on the block size to optimize each step in our GPU-based PBSN algorithm. The routines $ComputeRowMin()$ and $ComputeMin()$ are used to compare each value in the first half of a block with its corresponding value in the second half of the block, and stores the minimum. Similarly, the routines $ComputeRowMax()$ and $ComputeMax()$ are used to store the maximum values. If the block size is less than the width of the texture (as shown in the left), we compute the minimum values by rendering a quad of height H and block size width W (shown in grey color). Similarly, the maximum values are computed by rendering orange-colored blocks. If the block size is greater than the width of the texture (shown in the right), we compute the minimum values using the routine $ComputeMin()$ on the grey-colored quads. The pseudo-code for our algorithm is given in Routine 4.4.***

The output of one step is used as an input to the subsequent step. In each step, the algorithm decomposes the input into blocks of equal sizes, and performs the same set of comparison operations on each block. If the block size is B, a data value at the i-th location in the block is compared against an element at $(B - i)$-th location in the block. If $i \geq \frac{B}{2}$, the maximum of two elements being compared is placed at the i-th location, and if $i < \frac{B}{2}$, the minimum is placed at the i-th location.

The overall algorithm proceeds in $\log n$ stages, and during each stage $\log n$ steps are performed. At the beginning of each stage, the block size is initially set to n and a step of the algorithm is performed. At the end of each step, the block size is halved. At the end of $\log n$ stages, the input is sorted [16]. The case where n is a non-power of 2 is trivially handled by rounding n to the nearest power of 2 that is greater than n.

We map the PBSN algorithm to the GPU by using the blending and texture mapping functionalities of GPUs. The pseudo-code for our GPU-based algorithm is shown in Routine 4.3. Given an input sequence of length n, our algorithm stores the data values in a color component of a 2-D texture tex, and transfers tex to the GPU (line 1). The width and height of the texture are computed based on the value of $2^{\frac{logn}{2}}$ (line 2), though a more optimized implementation with tighter values is possible. Next we copy the data values into the frame buffer (line 3). At this stage, we use the blending operations to perform comparator operations in each stage of the algorithm. During each stage, we perform $\log n$ steps (line 5). Moreover as we proceed through different steps in a given stage, we vary the block size from n to 2 (line 6). The block size remains fixed while performing a single step. Based on the block size, we partition the input (implicitly) into several blocks and a Sortstep() routine is called on all the blocks in the input (line 7). The output of the Sortstep() routine is copied back to the input texture and used in the next step of the iteration (line 8). At the end of all the stages,

the data is readback to the CPU (line 11).

We improve the performance of our algorithm by optimizing the performance of the *Sortstep* routine. The pseudo-code for our Sortstep routine is shown in Routine 4.4. We have used two cases based on the block size and the width of the texture to optimize this routine. More details on these cases are shown in Fig. 2.

We further improve the performance of our algorithm by using the vector operations supported using the blending functionality of GPUs. Given an input sequence of length n, we store the data values in each of the four color components of the 2-D texture, and sort the four channels in parallel. The sorted sequences of length $\frac{n}{4}$ are readback by the CPU and a merge operation is performed in software. The merge routine performs $O(n)$ comparisons and is very efficient.

```
PBSN(n)
1  Compute and transfer a 2-D texture tex of size n to the GPU.
2  W = width(tex) = 2^⌊logn/2⌋, H = height(tex) = 2^⌈logn/2⌉
3  Copy(tex, W, H)
4  for i=1 to logn /* for each stage*/
5     for j=logn to 1
6        Block size B = 2^j
7        SortStep(tex, W, H, B);
8        Copy from frame buffer to tex
9     end for
10 end for
11 Readback sorted data to the CPU
```

ROUTINE 4.3: *Periodic Balanced Sorting Network Algorithm: This routine is used to sort an input sequence of length* n*. The input sequence is copied into a 2D-texture with width and height set to a power-of-2 that is closest to* $\sqrt{n}$ *(line 2). The texture is then copied into the frame buffer using the routine* $Copy()$*. Next, we perform* $\log n$ *stages on the input sequence and during each stage, perform* $\log n$ *steps with block sizes varying from* n *to* 2 *(line 6). Each step is performed using the routine SortStep() (line 8). At the end of each step, the data from the frame buffer is copied back into the input texture (line 8), and is used as input for the next step. At the end of all the stages, the sorted data is read back to the CPU (line 11). To improve the performance, we store the input sequence into each of the color channels of the texture and sort these sequences of length* $\frac{n}{4}$ *in parallel. A CPU-based merge routine is applied to combine the four sorted sequences.*

4.5 Analysis and Comparison

In this section, we analyze the performance of our algorithm and compare its performance with prior GPU-based and CPU-based sorting algorithms. Our algorithm performs $4 \times (\frac{n}{4})log^2(\frac{n}{4})$ comparisons on GPUs to sort four sequences of length $n/4$, and the merge operation performs n comparison operations on the CPU. Overall, our algorithm performs a total of $(n + nlog^2(\frac{n}{4}))$ comparisons to sort a sequence of length n. We exploit the inherent parallelism on GPUs and high-memory bandwidth to perform these operations.

We have compared the performance of our algorithm against optimized CPU-based sorting algorithms and a GPU-based sorting algorithm. We have benchmarked the performance of the algorithms on a PC with 3.4 GHz Intel Pentium IV CPU and a NVIDIA GeForce FX 6800 Ultra GPU. The performance of CPU-based algorithms was measured using two widely used compilers - Intel C++ compiler and Microsoft Visual C++ 6.0 compiler. We have enabled $-O2$ and $Qparallel$ flags to enable the SIMD optimizations. We used OpenGL to implement our GPU-based algorithms and optimized them using double buffered 16-bit offscreen buffers.

Fig. 3 highlights the performance of our algorithm (i.e. "our GPU algorithm") against Quicksort algorithm on the CPU and bitonic sort algorithm on the GPU [42]. For the Quicksort implementation, we have used the standard qsort routine available in $stdlib.h$.

```
SortStep(tex, width, height, blocksize)
1  if blocksize ≤ width
2     numrowblocks = width / blocksize
3     for i= 0 to (numrowblocks-1) /* for each row block*/
4        offset = i*blocksize
5        ComputeRowMin(tex, offset, blocksize, height)
6        ComputeRowMax(tex, offset, blocksize, height)
7  else
8     numblocks = width*height / blocksize, block_height = blocksize / width
9     for i= 0 to (numblocks-1) /* for each block*/
10       offset = i *block_height
11       ComputeMin(tex, offset, width, block_height)
12       ComputeMax(tex, offset, width, block_height)
13    end for
14 end if
```

ROUTINE 4.4: *This routine represents one step in our GPU-based sorting network algorithm. During each step, an element in the block at a position* k *is compared against an element in the same block at position* $blocksize - k$*. If* $k < \frac{blocksize}{2}$*, the minimum is stored in the element's location. This operation is performed on all the elements in the first half of a block and these elements correspond to the fragments in a quad of half the block size (indicated using orange and grey colors in Fig. 2). We use the routines* $ComputeMin()$ *and* $ComputeRowMin()$ *(lines 5 and 11) to perform the operation. These routines set the blend function to compute the minimum and render half-block sized quads with appropriate texture co-ordinates. For more details, refer to Routine 4.2. Similarly, if* $k > \frac{blocksize}{2}$*, the maximum is stored and we use the routines* $ComputeMax()$ *and* $ComputeRowMax()$ *on these elements (lines 6 and 12). Each routine is performed on every block (lines 3 and 9). We take advantage of the data layout in the texture and compute the minimum and maximum values efficiently on blocks with* $blocksize < width$ *of the texture. If the* $blocksize < width$*, then* $ComputeRowMin()$ *and* $ComputeRowMax()$ *routines render only a few quads as shown in Fig. 2. Each quad has width* $\frac{blocksize}{2}$ *and height* H*.*

For the bitonic sort implementation on the GPU, we have hand-optimized the source code provided by the original author as part of GPUGems[1], a collection of many different GPU programming tools.

The implementation of Quicksort in the Intel compiler has been optimized using Hyper-Threading technology. More details on the implementation of Quicksort are given here[2]. In particular, Quicksort can map well to a Hyper-Threading technology machine. The timings are from a parallelized implementation of Quicksort on the CPU and it balances the algorithm for the threaded scenario. This particular implementation also includes many other optimizations.

In our benchmarks, we have measured the performance of these algorithms on a random database with varying sizes of up to 8 million elements. The timings for GPU-based algorithm also include the time to transfer and readback the data along with the time to perform the sorting. The graph indicates that our GPU-based sorting algorithm outperforms the earlier CPU-based and the GPU-based implementations for reasonably large values of n. The graph also indicates that the Quicksort routine in the Intel compiler is well optimized and its performance is comparable to our GPU-based algorithm.

Fig. 4 illustrates the runtime behavior of our GPU-based sorting algorithm more explicitly. We have accounted for the data transfer time and computed the time taken to perform sorting on GPUs. In order to observe the $O(lg^2 n)$ behavior, we used a input size of 8M as the base reference for n and estimated the time taken to sort the remaining data sizes. The estimated points are highlighted

[1] http://developer.nvidia.com/object/all_tools_by_date.html

[2] http://www.intel.com/cd/ids/developer/asmo-na/eng/20372.htm?prn=Y

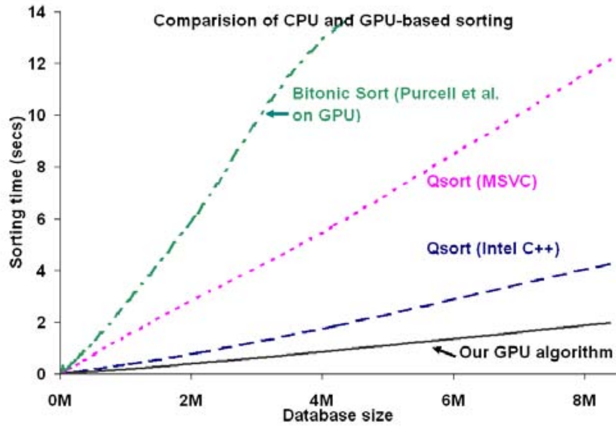

Figure 3: ***This graph compares the performance of our novel GPU-based sorting with an earlier GPU-based sorting algorithm [40] and Quicksort running on CPUs. We highlight two CPU-based implementations, based on two different compilers. In particular, the Intel compiler uses Hyper-Threading technology and optimizations to get very improved performance over standard implementations of Quicksort.***

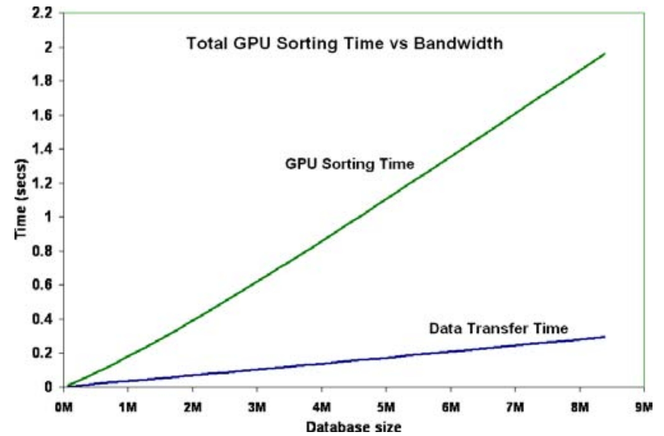

Figure 4: ***This graph shows the breakdown in our GPU-based sorting algorithm between the comparison operations on the GPU and data transfer time between the CPU and GPU. The sorting time is much higher than data transfer time. As a result, the bandwidth between CPU and GPU is not a bottleneck in our algorithm and implementation.***

in color yellow, and closely match the observed timings (within a few milli-seconds of accuracy). Our observations also indicate that the performance of our algorithm is around 3 times slower than optimized CPU-based Quicksort for small values of n ($n < 16K$). We attribute the slowdown to the constant overhead in the setup costs for our routine, which can dominate the overall time if the sorting time is low. Fig. 4 also shows that the data transfer times are not significant in comparison to the time spent in performing comparisons and sorting.

We have also analytically computed the number of clock cycles required to perform the sort operation, and used it to estimate the number of clock cycles required for each blending operation. We observed that the GPU requires $6-7$ clock cycles to perform one blending operation. In comparison, the earlier GPU-based bitonic sort algorithm [42] performs at least 53 instructions per pixel during each stage of the algorithm. Each instruction requires at least one clock cycle to execute and therefore, the GPU-based bitonic sort implementation [42] requires at least 53 instructions to execute a comparator stage. As our algorithm executes much fewer instructions, it is nearly an order of magnitude faster than prior GPU-based bitonic sort implementations. The performance of our algorithm is purely based on the performance of the underlying rasterization hardware, and is improving at a rate faster than the Moore's law for CPUs. Moreover, the architectural designs of GPUs can be further improved to perform faster blending. As a result, we expect that the performance gap between our GPU-based sorting algorithm and current CPU-based algorithms would increase on future generations of GPUs and CPUs.

5. PERFORMANCE AND APPLICATION

We have used our GPU-based sorting algorithm for frequency and quantile estimation in data streams. Specifically, we demonstrate its application to frequency estimation [34] and quantile estimation [22]. In this section, we present details of our implementation and compare its performance against optimized CPU implementations of quantile and frequency estimation algorithms. Finally, we show how our algorithm can be used with sliding windows. All the timings reported in this section were obtained on a PC with a 3.4 GHZ Pentium IV CPU with NVIDIA GeForce FX 6800 Ultra card, running Windows XP. We used the Intel compiler that provides an optimized implementation of Quicksort using Hyper-Threading Technology. The input data stream consists of 100 million elements with 16-bit floating point precision.

5.1 Frequency Estimation

ϵ-approximate frequency queries can be efficiently performed using histogram operations [27, 34]. Given a stream of data values, we use Manku and Motwani's [34] algorithm to compute an ϵ-approximate summary using limited space. The ϵ value is specified by the user. The ϵ-approximate summary data structure contains a subset of the elements of the stream along with an estimate of their frequency. Initially, the ϵ-approximate summary is set to empty. For each incoming window of size $\frac{1}{\epsilon}$, the algorithm computes a histogram using at most $\frac{1}{\epsilon}$ space. After that a merge operation is performed to insert or update the elements into the current ϵ-approximate summary. For each element in the histogram of the current window, the merge operation inserts the element and its frequency into the summary if the element does not exist in the summary. Otherwise, the frequency of the element is updated in the summary. A compress operation is then performed on the summary. In the compress operation, elements with a frequency of unity are deleted from the summary.

The resulting algorithm underestimates the frequencies of the elements in the summary by at most ϵN. Given a support s, the ϵ-approximate query returns all the elements in the ϵ-approximate summary with a frequency count of $(s-\epsilon)N$ as the output. The algorithm does not generate any false negatives and has a worst-case space requirement of $O(\frac{1}{\epsilon}log(\epsilon N))$.

The histogram operation is implemented space-efficiently using sorting. It turns out that 80-90% of the overall running time is spent in sorting. Fig. 6 shows the relative time spent in each of the three operations for various ϵ-values on a random database of 100 million elements. Fig. 5 compares the performance of our GPU-based algorithm against an optimized CPU-based implementation. The graph indicates that the GPU-based algorithm performs comparably to the CPU-based implementation on most benchmarks. Moreover, the performance of the GPU-based algorithm is better for larger window sizes, and the GPU incurs overhead for small window sizes.

5.2 Quantile Estimation

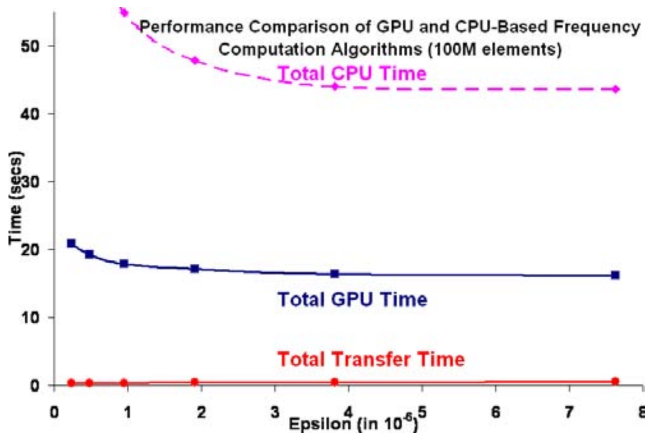

Figure 5: ***Performance comparison for frequency computations: This graph highlights the performance of our GPU-based algorithm against a CPU-based implementation of a frequency estimation algorithm. We have measured the performance on a random database of*** 100 ***million elements and used different*** ϵ ***values. The graph indicates that our GPU-based algorithm performs better than the optimized CPU implementation for large sized windows. The GPU-time also includes the time spent in the data transfer to and from the GPU. Moreover, the graph indicates that the data transfer time remains constant and is significantly lower than the time taken to sort the elements in the entire window.***

Given a large data stream of size N, where N is known a priori, the goal of quantile estimation algorithms is to efficiently compute ϵ-approximate quantiles using a limited memory footprint. Our algorithms is based on the approach described by Greenwald and Khanna [22]. In particular, Greenwald and Khanna proposed an efficient quantile estimation algorithm for sensor networks and compute an ϵ-approximate quantile summary while minimizing the communication costs in a sensor network. The sensor network is assumed as a tree with height h. Each node in the tree initially computes an $\frac{\epsilon}{2}$- approximate quantile summary by sorting its set of observations S locally, and choosing the elements of rank $1, \epsilon S, ..., |S|$. The summary structure also maintains the minimum rank and maximum rank for each element. Each node communicates its summary structure to its parent node. At the parent node, a merge operation is performed on these summaries. The merge operation sorts the union of these summaries and also updates the ranks of the elements using simple rules. Finally, the node performs a compress operation to compute a new summary structure with $B + 1$ elements, $B = \frac{h}{\epsilon}$. The new summary structure is computed by querying the current summary structure for elements of rank $1, \frac{|S|}{B}, \frac{2|S|}{B}, ..., |S|$. The new summary structure is $(\frac{\epsilon}{2} + \frac{i}{2B})$-approximate, where i is the height of the current node measured from the leaf in the tree. The compress operation reduces the size of the summary structure and is essential for space-efficient computation.

We extend the sensor network model in [22] to a stream model by maintaining the summary structure as an exponential histogram. Exponential histograms have been widely used for other statistic computations over sliding windows such as sums [13]. The exponential histogram consists of $\lceil logn \rceil$ buckets and each bucket is associated with a bucket id. Furthermore, each bucket stores a summary structure with an error based on its bucket id. If the bucket id is b, the error is set to $\frac{\epsilon}{2} + \frac{\epsilon b}{\lceil logn \rceil + 1}$. Initially, we set all the buckets as empty. Next, we compute an $\frac{\epsilon}{2}$-approximate summary for each new window of elements and assign it a bucket id of one and add it to the exponential histogram. If there are two buckets with same bucket id, we combine the two into one larger bucket and increment their bucket id by one. The combine operation involves a merge and prune operation performed using an error parameter. These operations are repeatedly performed on the exponential histogram till there are no two buckets with the same bucket id. Our overall algorithm has the same memory bound as [22]. It turns out that sorting takes a major fraction of running time (85-90%) in this algorithm. Fig. 7 shows the performance comparison of our algorithm using the qsort() and GPU-based sorting routines. The graph highlights the relative performance of our GPU-based implementation with an optimized CPU-based implementation. We observe that the performance of the GPU is comparable to the CPU in these benchmarks. For low window sizes, the performance of the CPU-based algorithm is better. This is mainly due to the fact that the elements in the window fit within the $L2$ cache on the CPU.

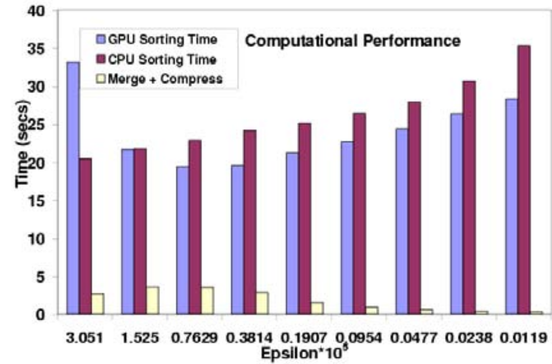

Figure 6: ***Cost of summary operations: In this graph, we show the time spent in performing each operation of our*** ϵ***-approximate summary computation algorithm for frequencies. The graph indicates that the majority of the computational time is spenting in sorting the window values.***

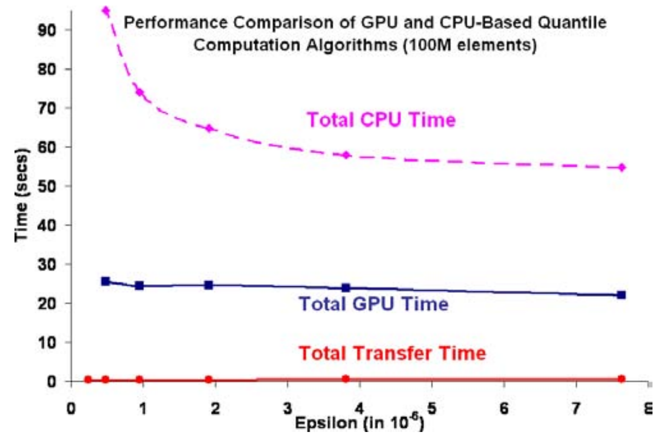

Figure 7: ***Performance comparison on quantile computations: This graph highlights the performance of our GPU-based implementation against a CPU-based implementation for quantile computations. The benchmarks are performed on a random database of*** 100***M elements with varying*** ϵ ***values. We observe that the GPU performance is comparable to a high-end Pentium IV CPU.***

5.3 Sliding Windows

We have applied our deterministic frequency and quantile estimation algorithms for performing ϵ-approximate queries over sliding windows. Given an incoming stream of elements, our goal is to perform ϵ-approximate queries over the last N elements. The

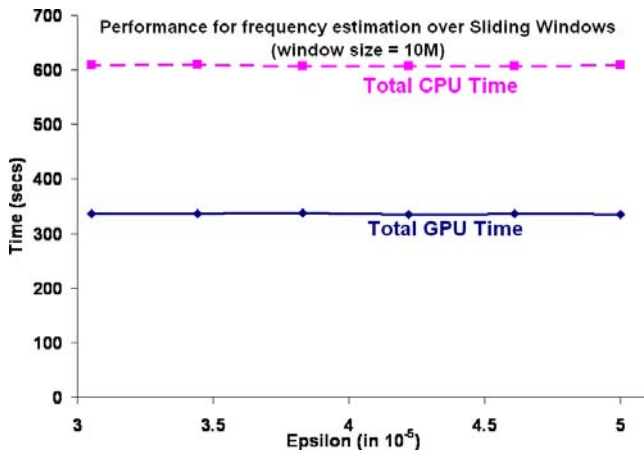

Figure 8: ***Performance of frequencies in sliding windows: This graph highlights the performance of our GPU-based implementation against a CPU-based implementation for frequency computations over a sliding window of size 1 million. The benchmarks are performed on a random database of*** 100***M elements with varying*** ϵ ***values. We have used the algorithm described by Arasu and Manku*** [6]***. In this algorithm, the*** ϵ ***values can be very large. This leads to a large number of small summary sizes for blocks at the bottom levels. The GPU-based algorithm is useful when we are computing large histogram sizes. The graph indicates that the performance of a GPU-based algorithm is comparable to that of a conventional CPU-based algorithm.***

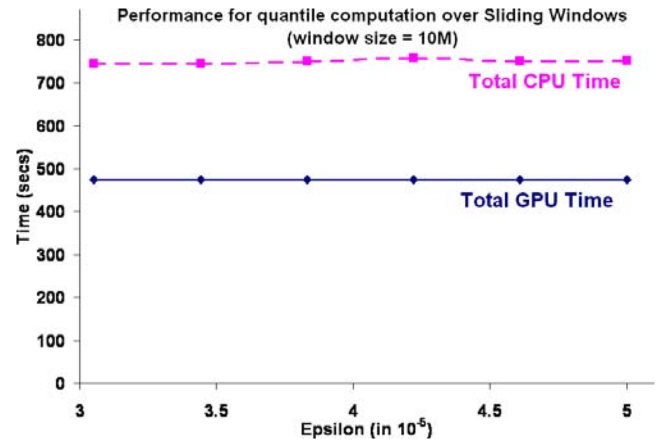

Figure 9: ***Performance of quantile computation in sliding windows of size 10 M elements***

window size N may be fixed or varying. Our algorithms are based on the recent work of Arasu and Manku [6]. Given an ϵ, our algorithm computes a deterministic bounded window-sketch with a limited memory footprint. The sketch maintains different levels, and the number of levels is $L = log\frac{4}{\epsilon}$. Furthermore, each level l uses an error-parameter $\epsilon_l = \frac{\epsilon}{2(2L+2)}2^{L-l}$ and computes an ϵ_l-approximate summary structure. We have used our algorithms described in Sections 5.1 and 5.2 to compute the ϵ_l-approximate summary.

Fig. 8 shows the performance of our implementation for estimating frequencies in a sliding window of 10 M elements. The algorithm proposed by Arasu and Manku [6] can lead to large epsilon values at the bottom (For e.g., $\epsilon_0 = \frac{1}{L+1}$ at level 0 and for an input epsilon of $4 * 10^{-6}$, $L \sim 20$ and $\epsilon_0 = 0.05$). It is inefficient to sort such small sets of values due to large error values on GPUs. Therefore, we have applied the GPU-based sorting algorithm only on the higher levels and the CPU-based quicksort elsewhere. The timings indicate a comparable performance using this implementation and a pure CPU-based implementation.

5.4 Related Applications

Our GPU-based sorting algorithm is also applicable to several other streaming algorithms. In such cases, we can use the GPU as a co-processor for sorting and distribute the load between the CPU and the GPU. The applications include:

- **Hierarchical Heavy Hitters:** Given a stream of size N, an error-parameter ϵ and a support s, a hierarchical heavy hitter (HHH) estimation query returns all the prefixes that occur in more than $(s-\epsilon)N$ elements of the stream. The query is useful for network failure analysis, denial-of-service attacks, etc. We can use our algorithm described in Section 5.1 to maintain an ϵ-approximate summary structure at each level of the prefix tree. Cormode et al. [12] proposed an efficient algorithm for the computation of HHHs by maintaining a trie-data structure. The trie-data structure maintains the prefixes of the elements as well as upper and lower bounds on the frequencies of elements with each prefix. The trie structure is very similar to the summary structure maintained by Greenwald and Khanna for quantile computation [23]. We use a slight variation of our algorithm presented in Section 5.2 to improve the performance of this algorithm.

- **Correlated Sums:** Correlated sum aggregate queries are a class of queries formed by applying correlated queries on pairs (x, y). They are of the form $SUM(g(y) : x \leq f(AGG(x))$, where AGG(x) is any basic aggregate query applied on input elements of the stream and $f(), g()$ are user-specified functions. These class of queries arise in network management and financial trading applications. Ananthakrishna et al. [4] proposed a space-efficient correlated sum estimation queries using two variations of the Greenwald and Khanna's algorithm for quantile estimation [23]. We can improve the performance of these queries by using our quantile estimation algorithm (described in Section 5.2).

6. ANALYSIS AND LIMITATIONS

In this section, we analyze the performance of our GPU-based sorting and numerical statistics computation algorithms. We identify different factors that govern the performance of our GPU-based streaming algorithms. We also highlight some GPU architectural features that can further improve the performance of our algorithms. These include:

- **Input data size:** The performance of our algorithm is highly dependent upon the size of the dataset used by the GPU-based sorting algorithm. In particular, the algorithm has a fixed amount of overhead in terms of the setup cost. This overhead can be significant when we are dealing with relatively small input data. Our timings indicate that the sorting performance of NVIDIA GeForce FX 6800 Ultra GPU is comparable or is slightly better than the optimized qsort routine on a 3.4 GHz Intel Pentium IV processor for data sizes $> 65K$.

- **Bus Bandwidth:** The available bus bandwidth between the GPU and the CPU governs the data transfer time. Our timings were collected on a machine with AGP-8x bus bandwidth to transmit the data from the CPU to the GPU. Recently, motherboards with PCI-E connectivity between the

CPU and GPU are becoming available and this will improve the bandwidth in both the directions. For large data sizes ($> 40KB$), we have observed that the data transfer time is not the dominating cost.

- **Blending and Texture Mapping:** The performance of the GPU-based sorting routine is governed by the performance of the underlying rasterization hardware. The performance of blending and texture mapping hardware can be improved by using better architectural design. For example, the texture representations can be compressed to yield better memory bandwidth to the frame buffer. Compression can improve the performance of our GPU-based sorting algorithm as each stage of the sorting algorithm reorders data elements to obtain better bit-based coherence. This can increasing the compression factor. Improvements in the texture caching performance (e.g. through higher bandwidth) would also improve the performance of our algorithm.

- **Load Balancing:** Our timings indicate that the current implementation of our algorithm spends a significant amount of time in GPU-based routines and the CPU remains idle during this time. We can improve the performance of our algorithms further by using the GPU in conjunction with the CPU. A preliminary implementation shows that we can obtain a *speedup of two times* using the CPU-based Quicksort routine in conjunction with our GPU-based sorting algorithm. We believe such hybrid approaches that utilize the CPU and the GPU simultaneously can offer significant speedups.

6.1 Limitations

Our current algorithm and implementation suffers from two main limitations.

- **Precision:** The precision of our sorting algorithm is limited to the precision of the underlying blending hardware on the GPUs. On current GPUs, only 8-bit and 16-bit floating point blending modes are supported. There has been considerable interest in supporting higher precision for blending and other special effects [3]. We expect that the future generations of GPUs will support higher precision blending operations.

- **Limited Memory:** GPUs support a limited amount of video memory and on current GPUs it is limited to 512MB. This limits the size of input data that our GPU-based algorithm can work on. As a result, we can handle lists with up to 32 million values. A possible solution to overcome this problem is to maintain an exponential histogram on the CPU, as described in Section 5.2. Databases in chunks of up to 32 million values can be streamed and sorted on the GPU, and a merge operation can be repeatedly performed in parallel on the CPU to sort the entire database efficiently.

7. CONCLUSIONS AND FUTURE WORK

We have presented novel GPU-based algorithms for numerical statistics computation on data streams. Our algorithms exploit the inherent parallelism and high memory bandwidth of GPUs for sorting. We use periodic balanced sorting network and perform the comparison and mapping operations by utilizing the blending and texture mapping capabilities of the GPUs. In practice, our GPU-based sorting algorithm is almost one order of magnitude faster than prior GPU-based sorting algorithm and comparable to one of the fastest CPU-based implementation of Quicksort. We have used our algorithm for frequency and quantile estimation over fixed and sliding windows. Overall, we show that GPU can be an effective co-processor for mining data streams. As the computational and rasterization performance of GPUs increases at a rate faster than the Moore's Law, our sorting and numerical statistics computation algorithm can be applied to more complex data streams.

There are many avenues for future work. We expect that the future GPUs will support higher precision for blending and we would like to use our algorithm for different applications. We would like to implement a sorting network based on bitonic sort and further improve the performance by developing cache-efficient GPU sorting algorithms [21]. The underlying sorting algorithm could also be useful for many other applications, including other numerical statistics and multidimensional histogram computation on continuous streams [24]. Recently, a new computational model has been proposed that augments the streaming model with a sorting primitive [3]. Our GPU-based sorting algorithm can be used for an efficient implementation of the algorithms based on this computational model on commodity PCs and laptops.

[3]http://www.nvidia.com/object/feature_HPeffects.html

Acknowledgements

Our work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134, 982167 and 0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, DARPA Contract N61339-04-C-0043 and Intel. We would like to thank NVIDIA corporation for their hardware and driver support.

References

[1] P. Agarwal, S. Krishnan, N. Mustafa, and S. Venkatasubramanian. Streaming geometric optimization using graphics hardware. *11 European Symposium on Algorithms*, 2003.

[2] 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.

[3] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. *Proc. of FOCS*, 2004.

[4] R. Ananthakrishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Srivastava. Efficient approximation of correlated sums on data streams. *IEEE Trans. on Knowledge and Data Engineering*, pages 569–572, 2003.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stanford data stream management system. *Data Stream Management*, 2004. To appear.

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

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

[8] K. E. Batcher. Sorting networks and their applications. In *Proc. 1968 Spring Joint Computer Conf.*, pages 307–314, Reston, VA, 1968. AFIPS Press.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computation on graphics hardware. *Proc. of ACM SIGGRAPH*, 2004.

[10] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Proc. of the 29th ICALP*, 2002.

[11] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Random sampling for histogram construction: How much is enough? *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):436–447, June 1998.

[12] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In ACM, editor, *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2003: San Diego, Calif., June 9–11, 2003*, pages 296–306, New York, NY 10036, USA, 2003. ACM Press. ACM order number 475030.

[13] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *Proc. of 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[14] E. Demaine, A. Lopez-Ortiz, and J. Munro. Frequency estimation of internet packet streams with limited space. *Proc. of 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.

[15] Michael Doggett. Programmability features of graphics hardware. *ACM SIGGRAPH Course Notes # 11*, 2003.

[16] M. Dowd, Y. Perl, L. Rudolpg, and M. Saks. The periodic balanced sorting network. *Journal of the ACM*, pages 738–757, 1989.

[17] Alain Fournier and Donald Fussell. On the power of the frame buffer. *ACM Transactions on Graphics*, 7(2):103–128, 1988.

[18] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. *Proc. of NSF Workshop on Next Generation Data Mining*, 2002.

[19] N. Govindaraju, M. Henson, M. Lin, and D. Manocha. Interactive visibility ordering of geometric primitives in complex environments. *Proc. of ACM Interactive 3D Graphics and Games*, 2005.

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

[21] N. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina-Chapel Hill, 2005.

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

[23] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):58–66, June 2001.

[24] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. *Proc. of the 18th International Conference on Data Engineering*, 2002.

[25] P. Indyk. Algorithms for dynamic geometric problems over data streams. *Proc. of STOC*, pages 373–380, 2004.

[26] C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. *Conference on Information and Knowledge Management*, 2003.

[27] R. Jin and G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. *Submitted for Publication*, 2004.

[28] R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. on Database Systems*, 28(1):51–55, 2003.

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

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

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

[32] A. Lastra, M. Lin, and D. Manocha. GPGP: General purpose computation using graphics processors. *http://www.cs.unc.edu/Events/Conferences/GP2*, 2004.

[33] X. Lin, H. Lu, J. Xu, and J. X. Xu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. *Proc. of 20th Int. Conf. on Data Engineering*, 2004.

[34] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In Philip A. Bernstein et al., editors, *VLDB 2002: proceedings of the Twenty-Eighth International Conference on Very Large Data Bases, Hong Kong*, pages 346–357, 2002.

[35] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):426–435, June 1998.

[36] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28(2):251–262, 1999.

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

[38] J. Misra and D. Gries. Finding repeated elements. *Sc. Comp. Prog.*, 2:143–152, 1982.

[39] R. Motwani and D. Thomas. Caching queues in memory buffers. *Proc. of SODA*, 2004.

[40] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, pages 315–323, 1980.

[41] S. Muthukrishnan. Data streams: Algorithms and applications. *Proc. of 14th ACM-SIAM Symposium on Discrete Algorithms*, 2003. http://athos.rutgers.edu/ muthu/stream-1-1.ps.

[42] 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.

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

[44] N. Thaper. Dynamic multidimensional histograms. *Proc. of ACM SIGMOD*, 2002.

[45] S. Venkatasubramanian. The graphics card as a stream computer. *Workshop on Management and Processing of Data Streams*, 2003.

[46] J. Xu, X. Lin, and X. Zhou. Space efficient quantile summary for constrained sliding windows on a data stream. *Proceedings of WAIM (LNCS 3129)*, pages 34–44, 2004.

[47] 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.