GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management

Naga K. Govindaraju [*] Jim Gray [†] Ritesh Kumar [*] Dinesh Manocha [*]

{naga,ritesh,dm}@cs.unc.edu, Jim.Gray@microsoft.com
http://gamma.cs.unc.edu/GPUTERASORT

ABSTRACT

We present a new algorithm, GPUTeraSort, to sort billion-record wide-key databases using a graphics processing unit (GPU) Our algorithm uses the data and task parallelism on the GPU to perform memory-intensive and compute-intensive tasks while the CPU is used to perform I/O and resource management. We therefore exploit both the high-bandwidth GPU memory interface and the lower-bandwidth CPU main memory interface and achieve higher memory bandwidth than purely CPU-based algorithms. GPUTeraSort is a two-phase task pipeline: (1) read disk, build keys, sort using the GPU, generate runs, write disk, and (2) read, merge, write. It also pipelines disk transfers and achieves near-peak I/O performance. We have tested the performance of GPUTeraSort on billion-record files using the standard Sort benchmark. In practice, a 3 GHz Pentium IV PC with $265 NVIDIA 7800 GT GPU is significantly faster than optimized CPU-based algorithms on much faster processors, sorting 60GB for a penny; the best reported PennySort price-performance. These results suggest that a GPU co-processor can significantly improve performance on large data processing tasks.

1. INTRODUCTION

Huge sort tasks arise in many different applications including web indexing engines, geographic information systems, data mining, and supercomputing. Sorting is also a proxy for any sequential I/O intensive database workload. This article considers the problem of sorting very large datasets consisting of billions of records with wide keys.

The problem of external memory sorting has been studied for more than five decades, starting with Friend [16]. The dramatic improvements in the speed of sorting algorithms are largely due to advances in computer architecture and software parallelism. Recent algorithms utilize simultaneous multi-threading, symmetric multi-processors, advanced memory units, and multi-processors to improve sorting performance. The current Indy PennySort record benchmark[1], sorts a 40 GB database in 1541 seconds on a $614 Linux/AMD system.

However, current external memory sort performance is limited by the traditional Von Neumann style architecture of the CPU. Computer architects use data caches to ameliorate the CPU and the main memory bottleneck; but, CPU-based sorting algorithms incur significant cache misses on large datasets.

This article shows how to use a commodity graphics processing unit (GPU) as a co-processor to sort large datasets. GPUs are programmable parallel architectures designed for real-time rasterization of geometric primitives - but they are also highly parallel vector co-processors. Current GPUs have 10x higher main memory bandwidth and use data parallelism to achieve 10x more operations per second than CPUs. Furthermore, GPU performance has improved faster than Moore's Law over the last decade - so the GPU-CPU performance gap is widening. GPUs have recently been used for different scientific, geometric and database applications, as well as in-memory sorting [20, 22, 35]. However, previous GPU-based sorting algorithms were not able to handle gigabyte-sized databases with wide keys and could not keep up with modern disk IO systems.

Main Results: We present GPUTeraSort that uses a GPU as a co-processor to sort databases with billions of records. Our algorithm is general and can handle long records with wide keys. This hybrid sorting architecture offloads compute-intensive and memory-intensive tasks to the GPU to achieve higher I/O performance and better main memory performance. We map a bitonic sorting network to GPU rasterization operations and use the GPU's programmable hardware and high bandwidth memory interface. Our novel data representation improves GPU cache efficiency and minimizes data transfers between the CPU and the GPU. In practice, we achieve nearly 50 giga-byte per second memory bandwidth and 14 giga-operations per second on a current GPU. These numbers are 10x what we can achieve on the CPU.

We implemented GPUTeraSort on an inexpensive 3 GHz Pentium IV EE CPU with a $265 NVIDIA 7800 GT GPU. GPUTeraSort running the SortBenchmark on this inexpensive computer has performance comparable to an "expensive" $2,200 3.6 GHz Dual Xeon server. Our experimental results show a 4 times performance improvement over the 2005 Daytona PennySort benchmark record and 1.4 times

[*]University of North Carolina at Chapel Hill
[†]Microsoft Research

[1]http://research.microsoft.com/barc/SortBenchmark

improvement over the 2003 Indy PennySort benchmark record. Some of the novel contributions of our work include:

- An external sorting architecture that distributes the work between the CPU and GPU.
- An in-memory GPU-based sorting algorithm which is up to 10 times faster than prior CPU-based and GPU-based in-memory sorting algorithms.
- Peak I/O performance on an inexpensive PC and near peak memory bandwidth on the GPU.
- A scalable approach to sorting massive databases by efficiently sorting large data partitions.

In combination, these features allow an inexpensive PC with a mid-range GPU to outperform much more expensive CPU-only PennySort systems. The rest of the paper is organized as follows. Section 2 reviews related work on sorting, hardware accelerated database queries, and GPU-based algorithms. Section 3 highlights some of the limitations of CPU-based external sorting algorithms and gives an overview of GPUTeraSort. Section 4 presents the GPUTeraSort algorithm and Section 5 describes its implementation. Section 6 compares its performance with prior CPU-based algorithms.

2. RELATED WORK

This section briefly surveys related work in sorting and the use of GPUs to accelerate data management computations.

2.1 Sorting

Sorting is a key problem in database and scientific applications. It has also been well studied in the theory of algorithms [23]. Many optimized sorting algorithms, such as quicksort, are widely available and many variants have been described in the database literature [2]. However, the CPU performance of sorting algorithms is governed by cache misses [17, 24, 32] and instruction dependencies [45]. To address these memory and CPU limits, many parallel algorithms and sorting systems have been proposed in the database and high performance computing literature [11, 14, 25, 38, 44].

The Sort Benchmark, introduced in 1985 was commonly used to evaluate the sorting algorithms [15]. As the original benchmark became trivial, it evolved to the MinuteSort [32] and the PennySort benchmarks [33]. Nyberg et al. [32] use a combination of quicksort and selection-tree mergesort in the AlphaSort algorithm. In practice, AlphaSort's performance varied considerably based on the cache sizes. The NOW-SORT algorithm [8] used a cluster of workstations to sort large databases. Recently, Garcia and Korth [17] used features of SMT (simultaneous multi-threading) to accelerate in-memory sort performance.

2.2 Optimizing Multi-Level Memory Accesses

Many algorithms have been proposed to improve the performance of database operations using multi-level memory hierarchies that include disks, main memories, and several levels of processor caches. Ailamaki gives a recent survey on these techniques [4]. Over the last few years, database architectures have used massive main memory to reduce or eliminate I/O; but the resulting applications still have very high clocks per instruction (CPI). Memory stalls due to cache misses can lead to increased query execution times [6, 27]. 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. These techniques can also improve the performance of sorting algorithms [5, 12, 26, 28, 36, 37, 39, 45].

2.3 GPUs and Data Parallelism

Many special processor architectures have been proposed that employ data parallelism for data intensive computations. Graphics processing units (GPUs) are common examples of this, but there are many others. The ClearSpeed CSX600 processor [1] is an embedded, low power, data parallel co-processor that provides up to 25 GFLOPS of floating point performance. The Physics Processing Unit (PPU) uses data parallelism and high memory bandwidth in order to achieve high throughput for Physical simulation. Many other co-processors accelerate performance through data parallelism.

This paper focuses on using a GPU as a co-processor for sorting, because GPUs are commodity processors. A high performance mid-range GPU costs less than $300. Current GPUs have about 10× the memory bandwidth and processing power of the CPU and this gap is widening. Commodity GPUs are increasingly used for different applications including numerical linear algebra, scientific, and geometric computations [34]. GPUs have also been used as co-processors to speedup database queries [9, 18, 19, 40] and data streaming [20, 29, 41].

Sorting on GPUs: Many researchers have proposed GPU-based sorting algorithms. Purcell et al. [35] describe a bitonic sort using a fragment program where each stage of the sorting algorithm is performed as one rendering pass. Kipfer et al. [22] improve bitonic sort by simplifying the fragment program; but the algorithm still requires ~ 10 fragment instructions. Govindaraju et al. [20] present a sorting algorithm based on a periodic balanced sorting network (PBSN) and use texture mapping and blending operations. However, prior GPU-based algorithms have certain *limitations* for large databases. These include:

- *Database size*: Previous algorithms were limited to databases that fit in GPU memory (i.e. 512MB on current GPUs).
- *Limit on key size*: The sort keys were limited to 32-bit floating point operands.
- *Efficiency*: Previous algorithms were not fast enough to match the disk array IO bandwidth.

Our GPUTeraSort algorithm uses the GPU as a co-processor in ways that overcome these limitations.

3. OVERVIEW

This section reviews external memory sorting algorithms, analyzing how these algorithms use processors, caches, memory interfaces, and input/output (I/O) devices. Then we present our GPUTeraSort algorithm.

3.1 External Memory Sorting

External memory sorting algorithms are used to reorganize large datasets. They typically perform two phases. The first phase produces a set of files; the second phase processes these files to produce a totally ordered permutation of the input data file. External memory sorting algorithms can be classified into two broad categories [42]:

- **Distribution-Based Sorting:** The first phase partitions the input data file using (S-1) partition keys and generates S disjoint buckets such that the elements in one bucket precede the elements in the remaining buckets [23]. In the second phase, each bucket is sorted independently. The concatenated sorted buckets are the output file.

- **Merge-Based Sorting:** The first phase partitions the input data into data chunks of approximately equal size, sorts these data chunks in main memory and writes the "runs" to disk. The second phase merges the runs in main memory and writes the sorted output to the disk.

External memory sorting performance is often limited by I/O performance. Disk I/O bandwidth is significantly lower than main memory bandwidth. Therefore, it is important to minimize the amount of data written to and read from disks. Large files will not fit in RAM so we must sort the data in at least two passes but two passes are enough to sort huge files. Each pass reads and writes to the disk. Hence, the two-pass sort throughput is at most $\frac{1}{4}$ the throughput of the disks. For example, a PC with 8 SATA disks each with a peak I/O bandwidth of 50 MBps per disk can achieve at most 400 MBps disk bandwidth. So a p-pass algorithm will have a throughput of $\frac{400}{2p}$ since each pass must read as well as write the data. In particular, a two-pass sort achieves at most 100 MBps throughput on this PC. Single pass algorithms only work on databases that fit entirely in main memory.

External memory sort algorithms can operate in two passes if the Phase 1 partitions fit in main memory. The parallel disk model (PDM) [43] captures disk system performance properties. PDM models the number of I/O operations, disk usage and CPU time. Vitter [42] analyzed the practical applicability of PDM model to common I/O operations such as scanning the items in a file, sorting a file, etc. In this model, the average and worst case I/O performance of external memory sorting algorithms is $\approx \frac{n}{D} log_m n$ where n is the input size, m is the internal memory size, D is the number of disks and $log_m n$ denotes the number of passes when the data partition size in the Phase 1 is $\approx m$ [3, 30]. Based on the PDM model, an external memory sorting algorithm can achieve good I/O performance on large databases when the data partition sizes are comparable to the main memory size. Salzberg et al. [38] present a similar analysis of merge based sorting memory requirements. The analysis is as follows. If N is the file size, M is the main memory size and R is the run size in phase 1 then typically: (1) $R \approx \frac{M}{3}$ because of the memory required to simultaneously pipeline reading the input, sorting, and writing the output. The number of runs generated in phase 1 is $runs \approx \frac{N}{R}$. If T is the I/O read size per run in phase 2, and then since at least one buffer for each run must fit in memory and a few more buffers are needed for prefetch and postwrite: (2) $M \approx T \times runs \approx T \times \frac{N}{R}$. Combining equations (1) and (2) gives (3) $M^2 \approx T \times \frac{N}{3}$ or, ignoring the constant term (4) $M \approx \sqrt{TN}$.

Since a two-pass sort's RAM requirements (M) increase as the square root of the input file size, multi-GB RAM machines can two-pass sort terabyte files. In particular, if T=2 MB to reduce disk seek overhead, and N is 100 GB, then $R \sim 230$ MB. In practice, phase 1 partitions are hundreds of megabytes on current PCs. However, current algorithms running on commodity CPUs, referred to as CPU-based algorithms, cannot achieve high sorting performance on such large partitions because:

- **Cache Misses:** CPU-based sorting algorithms incur significant cache misses on data sets that do not fit in the L1, L2 or L3 data caches [32]. Therefore, it is not efficient to sort partitions comparable to the size of main memory. This results in a tradeoff between disk I/O performance (as described above) and CPU computation time spent in sorting the partitions. For example, in merge-based external sorting algorithms, the time spent in Phase 1 can be reduced by choosing run sizes comparable to the CPU cache sizes. However, this choice increases the time spent in Phase 2 to merge a large number of small runs. Figure 1 illustrates the performance of an optimized commercial CPU based algorithm [31] on a dual Xeon configuration for varying Phase 1 run sizes. Observe that the elapsed time decreases as the run size increases. However, increasing the run size beyond the CPU data cache sizes can degrade the sorting performance during Phase 1 [24]. As explained in Section 4, GPUs have a high bandwidth memory interface that can achieve higher performance on larger runs.

- **I/O Performance:** I/O operations have relatively low CPU overhead. However, CPU-based sorting algorithms can be compute-intensive [24] and may not be able to achieve high I/O performance. Figure 13 highlights the I/O performance of Nsort [31] on systems with a peak I/O throughput of 200 MBps. The I/O throughput obtained by the CPU-based sorting algorithm is around 147 MBps for a single processor and

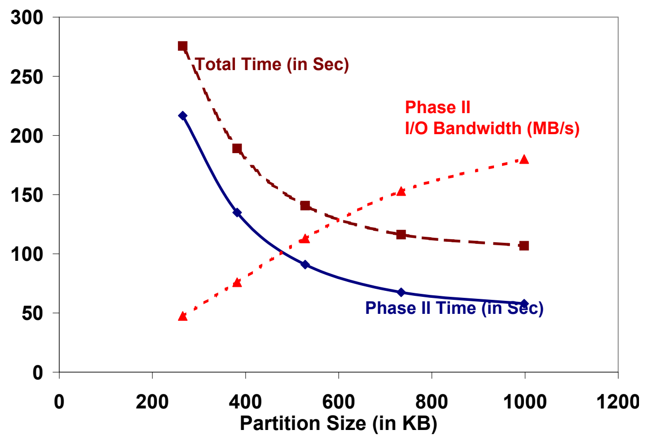

Figure 1: Performance of an optimized merge-based external memory sorting algorithm on a Dual 3.6 GHz Xeon processor system. Observe that the speed of Phase 2 increases nearly linearly with the partition size. As the data partition sizes in Phase I fit well in the L2 cache sizes, the Phase 1 time remains nearly constant.

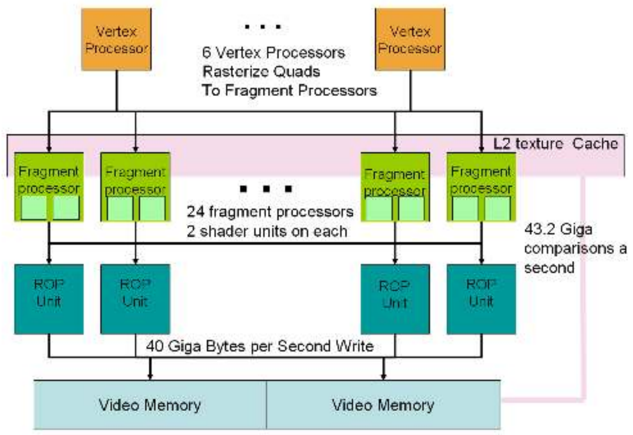

Figure 2: This figure highlights the high data parallelism and memory bandwidth inside a GPU. GPUTeraSort uses the vector processing functionalities to implement a highly parallel bitonic sorting network. It outperforms prior CPU-based and GPU-based algorithms by 3-10 times.

around 200 MBps with a dual processor. This suggests that the overall I/O performance can be improved by offloading computation to an additional processor or co-processor.

- **Memory Interfaces:** Some recent external sorting algorithms use simultaneous multi-threading (SMT) and chip multi-processor (CMP) architectures to improve performance. However, the interface to main memory on current SMT and CMP architectures significantly limits the memory bandwidth available to each thread when data does not fit in processor caches [17]. It is possible to achieve higher performance by running the sorting algorithm on co-processors with dedicated memory interfaces.

3.2 Sorting with a Graphics Processor

This section gives a brief overview of graphics processors (GPUs) highlighting features that make them useful for external memory sorting. GPUs are designed to execute geometric transformations on a rectangular pixel array. Each transformation generates a data stream of display pixels. Each incoming data element has a color and a set of texture coordinates that reference a 2D texture array. The data stream is processed by a user specified program executing on multiple fragment processors. The output is written to the memory. GPUs have the following capabilities useful for data-intensive computations.

- **Data Parallelism:** GPUs are highly data parallel - both partition parallelism and pipeline parallelism. They use many fragment processors for partition parallelism. Each fragment processor is a pipeline-parallel vector processor that performs four concurrent vector operations such as multiply-and-add (MAD) instructions on the texture coordinates or the color components of the incoming data stream. Current CPUs offer similar data parallelism using instructions such as SSE2 on Intel processors or AltiVec operations on PowerPC processors. However, CPU data parallelism is relatively modest by comparison. In case of sorting, a high-end Pentium IV processor can execute four SSE2 comparisons per clock cycle while a NVIDIA GeForce 7800 GTX GPU-based sorting algorithm can perform 96 comparisons per clock cycle.

- **Instruction-level Parallelism:** In addition to the SIMD and vector processing capabilities, each fragment processor can also exploit instruction-level parallelism, evaluating multiple instructions simultaneously using different ALUs. As a result, GPUs can achieve higher performance than CPUs. For example, the peak computational performance of a high-end dual core Pentium IV processor is 25.6 GFLOPS, whereas the peak performance of NVIDIA GeForce 7800 GTX is 313 GFLOPS. GPU instruction-level parallelism significantly improves sort performance, overlapping sort-key comparisons operations while fetching the pointers associated with the keys to achieve near-peak computational performance.

- **Dedicated Memory Interface:** The GPU's memory controller is designed for high bandwidth data streaming between main memory and the GPU's onboard memory. GPUs have a wider memory interface than the CPU. For example, current high-end PCs have 8-byte main memory interface with a peak memory bandwidth of 6.4 GB per second, whereas, a NVIDIA 7900 GTX has a 64-byte memory interface to the GPU video memory and can achieve a peak memory bandwidth of 56 GB per second.

- **Low Memory Latency:** GPUs have lower computational clock rates ($\sim 690MHz$) than memory clock rates ($\sim$ 1.8 GHz) but reduce the memory latency by accessing the data sequentially thereby allowing prefetch and pipelining. In contrast, CPUs have higher computational clock rates ($\sim$ 4 GHz) than main memory speeds ($\sim$ 533 MHz) but suffer from memory stalls both because the memory bandwidth is inadequate and because they lack a data-stream approach to data access.

Many GPU-based sorting algorithms have been designed to exploit one or more of these capabilities [20, 22, 35]. However, those algorithms do not handle large, wide-key databases and have other limitations, highlighted in Section 2.

In summary, GPUs offer 10× more memory bandwidth and processing power than CPUs; and this gap is widening. GPUs present an opportunity for anyone who can use them for tasks beyond graphics [34].

3.3 Hybrid Sorting Architecture

This section gives an overview of GPUTeraSort. The next section describes the use of the GPU in detail. Our goal is to design a sorting architecture to efficiently utilize the computational processors, I/O and memory resources. GPUTeraSort has five stages that can be executed sequentially; but, some stages can be executed using multi-buffered pipeline-parallel independent threads:

- **Reader:** The reader asynchronously reads the input file into a (approximately 100 MB) main memory buffer (zero-copy direct IO). Read bandwidth is improved by

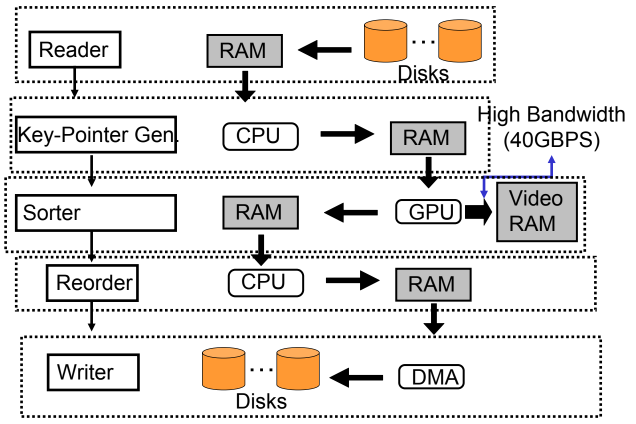

Figure 3: Flow Diagram of Phase 1 of GPUTeraSort Architecture using GPUs and CPUs.

striping the input file across all disks so the data is transferred from all disks in parallel. The I/O bandwidth and the CPU usage of the reader depend on the number of overlapping asynchronous I/O requests, the stripe size, and the number of disks in the stripe. The reader thread requires less than 10% of a CPU to achieve near-peak I/O performance.

- **Key-Generator:** The Key-Generator computes the (key, record-pointer) pairs from the input buffer. In practice, this stage is not computationally intensive but can be memory-intensive, reading each key from main memory. It sequentially writes a stream of key-pointer pairs to main memory.
- **Sorter:** The Sorter reads and sorts the key-pointer pairs. This stage is computationally intensive and memory-intensive on large buffers with wide keys (e.g. of size 10 bytes or more). For example, the throughput of an SSE-optimized CPU-based quicksort on a 3.4 GHz Pentium IV sorting 1 million floating point keys is much less than the throughput of the other external memory sorting stages and is the bottleneck. This is shown by Figure 11 and by the quicksort performance in Figure 8.
- **Reorder:** The reorder stage rearranges the input buffer based on the sorted key-pointer pairs to generate a sorted output buffer (a run). On large databases, reorder is expensive because it randomly reads and writes long records from the input buffer and so has many memory stalls (Figure 11).
- **Writer:** The writer asynchronously writes the run to the disk. Striping a run across many disks is not efficient for Phase 2 reads[42]; therefore GPUTerasStort cyclically writes the Phase 1 runs to individual disks in very large transfers. The writer thread requires less than 10% of the CPU to achieve near-peak I/O performance.

Figure 3 shows GPUTeraSort's pipeline flow. In order to efficiently pipeline these stages, GPUTeraSort uses a GPU as a co-processor to perform the key-pointer sorter task. The new sorting architecture

- Performs the key-pointer sorting on the GPU and frees CPU cycles to achieve higher I/O performance and throughput.
- Reduces the memory contention by using the dedicated GPU memory for sorting.

4. LARGE SORTS USING GPUS

This section describes GPUTeraSort's sorting algorithm to sort wide keys and pointers on GPUs using a novel data representation. The algorithm improves cache efficiency and minimizes data transfer overheads between the CPU and GPU. A theoretical and experimental analysis of GPUTeraSort's data transfer rate and memory bandwidth requirements compares the performance with prior algorithms.

4.1 Bitonic Sorting on GPUs

GPU-based algorithms perform computations on 2D arrays of 32-bit floating point data values known as *textures*. Each array element corresponds to a *pixel*. Pixels are transformed by programmable *fragment processors*, each executing the same *fragment program* on each pixel. the multiple GPU fragment processors perform data parallel computations on different pixel arrays simultaneously. This simple data-parallel architecture avoids write-after-read hazards while performing parallel computations.

At high-level GPU-based sorting algorithms read values from an input array or texture, perform data-independent comparisons using a fragment program, and write the output to another array. The output array is then swapped with the input array, and the comparisons are iteratively performed until the whole array is sorted. These sorting network algorithms map well to GPUs.

The *bitonic sorting network* [10] sorts bitonic sequences in multiple merge steps. A *bitonic sequence* is a monotonic ascending or descending sequence.

Given an input array $a = (a_0, a_1, \ldots, a_n)$, the bitonic sorting algorithm proceeds bottom-up, merging bitonic sequences of equal sizes at each stage. It first constructs bitonic sequences of size 2 by merging pairs of adjacent data elements (a_{2i}, a_{2i+1}) where $i = 0, 1, \ldots, \frac{n}{2} - 1$. Then bitonic sequences of size 4 are formed in stage 2 by merging pairs of bitonic sequences (a_{2i}, a_{2i+1}) and $(a_{2i+2}, a_{2i+3}), i = 0, 1, \ldots, \frac{n}{2} - 2$. The output of each stage is the input to the next stage. The size of the bitonic sequence pairs doubles at every stage. The final stage forms a sorted sequence by merging bitonic sequences $(a_0, a_1, ., a_{\frac{n}{2}}), (a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \ldots, a_n)$ (see Figure 4).

Specifically, stage k is used to merge two bitonic sequences, each of size 2^{k-1} and generates a new bitonic sequence of length 2^k. The overall algorithm requires $logn$ stages. In stage k, we perform k steps in the order k to 1. In each step, the input array is conceptually divided into chunks of equal sizes (size $d = 2^{j-1}$ for step j) and each elements in one chunk is compared against the corresponding element in its adjacent chunks i.e., an element a_i in a chunk is compared with the element at distance d (a_{i+d} or a_{i-d}). The minimum is stored in one data chunk and the maximum is stored in the other data chunk. Figure 4 shows a bitonic sorting network on 8 data values. Each data chunk in a step is color coded and elements in adjacent data chunks

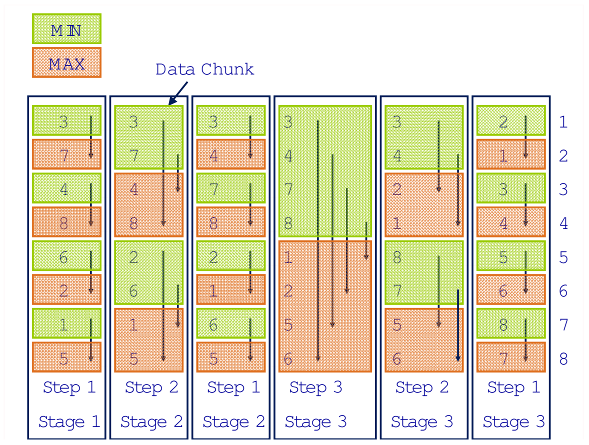

Figure 4: This figure illustrates a bitonic sorting network on 8 data values. The sorting algorithm proceeds in 3 stages. The output of each stage is the input to the next stage. In each stage, the array is conceptually divided into sorted data chunks or regions highlighted in green and red. Elements of adjacent chunks are merged as indicated by arrows. The minimum element is moved to the green region and the maximum is stored in the red colored regions producing larger sorted chunk.

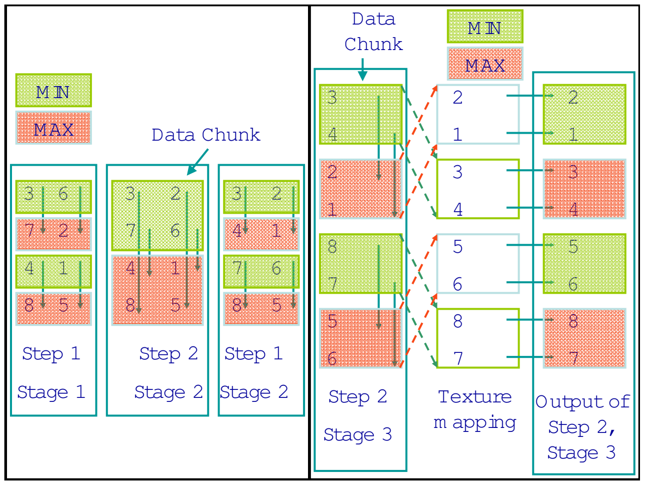

Figure 5: The left figure shows the 1-D mapping of comparisons among array elements in step 2 and stage 3 of Figure 4. The mapping is implemented using GPU texturing hardware. For each data chunk, we pass the element indices (or vertex locations of a 1-D line) of the corresponding data chunk for comparisons. The texturing hardware fetches the data values at the corresponding locations for each pixel, and a single-instruction fragment program computes the minimum or maximum in parallel on multiple pixels simultaneously using the fragment processors. The right figure shows the 2D-representation of the 1-D array of size 8 shown in Figure 4. In this example, the width of the 2D array is 2 and height is 4. Observe that the data chunks now correspond to row-aligned quads and the sorting network maps well to the GPU 2D texturing hardware.

are compared. The minimum is stored in the green colored region and the maximum is stored in the red colored region. For further details on the bitonic sorting algorithm refer to [13].

In a GPU, each bitonic sort step corresponds to mapping values from one chunk in the input texture to another chunk in the input texture using the GPU's texture mapping hardware as shown in Figure 5. The texture mapping hardware fetches data values at a fixed distance from the current pixel and compares against the current pixel value and may replace the value based on the comparison. The texturing hardware works as follows. First, a 2D array is specified to fetch the data values. Then, a 2D quadrilateral is specified with lookup co-ordinates for vertices. For every pixel in the 2D quadrilateral, the texturing hardware performs a bilinear interpolation of the lookup co-ordinates of the vertices. The interpolated coordinate is used to perform a 2D array lookup by the fragment processor. This results in the larger and smaller values being written to the higher and lower target pixels. The left Figure 5 illustrates the use of texture mapping for sorting. In this example, 1-D lines are specified for data chunks with appropriate lookup co-ordinates. For example, the first line segment $(0, 1)$ is specified with the vertex lookup co-ordinates $(2, 3)$. Then, the texture mapping hardware is used to directly fetch values a_2, a_3 and compare them against a_0 and a_1 respectively within the fragment processor.

As GPUs are primarily optimized for 2D arrays, we map the 1D array onto a 2D array as shown in Figure 5. The resulting data chunks are 2D data chunks that are either row-aligned (as shown in the right side of Figure 5) or column-aligned. The resulting algorithm maps well to GPUs. The pseudo-code for the algorithm is shown in Routine 4.1.

4.2 Improved Algorithm

GPUTeraSort uses an efficient data representation on GPUs to sort key-pointer pairs. Current GPUs support 32-bit floating point numbers. GPUTeraSort represents the bytes of the keys as 32-bit floating point 2D arrays (or textures). Each element in the texture is associated with four color channels - red, blue, green and alpha. Given a texture of width W and height H, there are two possible representations for designing a data-parallel bitonic sorting algorithm:

- **Single-array representation:** Each texture is represented as a stretched 2D array and each texel in the array is replaced with its four channels. Therefore, a texture of width W and height H can be represented as a stretched 2D array of width $4W$ and height H.

- **Four-array representation:** In this representation, each texture is composed of four subarrays and each subarray corresponds to a single channel. Therefore, a texture of width W and height H can be represented using four independent arrays of width W and H. These four arrays can be sorted simultaneously using SIMD instructions [20]. Finally, a merge operation is performed to generate the sorted array of key-pointers.

In each of these representations, the keys and the pointers associated with the keys are stored in two separate textures: a key texture and a pointer texture. The single-array

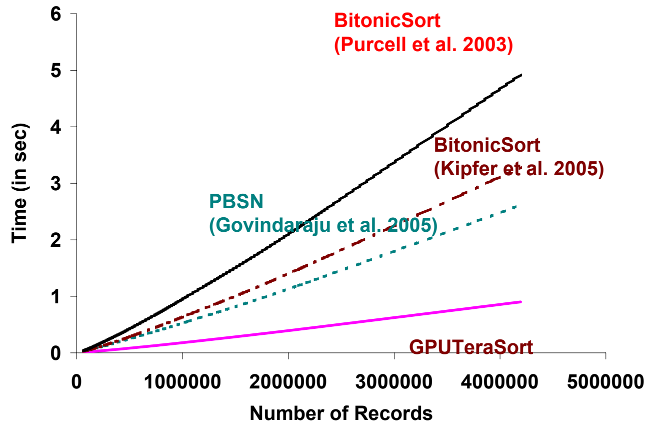

Figure 6: Comparison of GPUTeraSort's bitonic sort with other GPU-based algorithms indicates a 3-7 fold performance advantage over prior GPU-based bitonic sort and PBSN algorithms (Kipfer et al. [23], Govindaraju et al. [20], and Purcell et al. [36]).

```
BitonicSort(tex, W,H)
1  n = numValues to be sorted = W*H*4 /* single array representation*/
2  for i=1 to logn /* for each stage*/
3      for j=i to 1
4          Quad size B = $2^{j-1}$
5          Draw Textured Quads of size B
6          Copy from frame buffer to tex
7      end for
8  end for
```

ROUTINE 4.1: *Bitonic Sorting Network Algorithm: We use this routine to sort a floating point input sequence of length n. Next, we perform* $\log n$ *stages on the input sequence and during each stage, perform i steps with quad sizes (width* $\times$ *height) varying from* 2^{i-1} *to 1 (line 4). The overall algorithm requires* $O(\frac{nlg^2 n}{2})$ *comparisons and maps well to GPUs.*

representation, as opposed to the four-array representation proposed by Govindaraju et al. [20], has the following advantages:

- **Mapping:** The data transfer operation from the CPU to the GPU directly maps to the single-array representation while the four-array representation does not.
- **Efficient sorting:** The single-array representation has better performance than four-array representation as it reduces the memory accesses in early algorithm steps. For example, steps 1 and 2 of each stage in the algorithm compute bitonic sequences of size 4. In a single-array representation, the four values are stored in a single element of the texture, fetched using a single memory fetch and sorted within the registers of the fragment processor. This is not possible in the four-array representation.

Data is transferred to the GPU using a DMA on a dedicated PCI-Express bus at up to 4 GBps. To improve cache-efficiency, GPUTeraSort lowers memory latency by overlapping pointer and fragment processor memory accesses. This significantly improves performance.

```
GPUTeraSort(n)
1  b = number of bytes in key
2  $W = width(tex) = 2^{\lfloor \frac{logn}{2} \rfloor}$, $H = height(tex) = 2^{\lceil \frac{logn}{2} \rceil}$
3  sorted = false
4  currBytes = Most significant four bytes of keys
5  SortLists = Input Array of Key-Pointers
6  While (!sorted)
7    For each array in sortedLists
8        Transfer the pointerTexture and keyTexture of currBytes to the GPU
9        Perform HybridBitonicSort(pointerTexture,keyTexture,n)
10       Readback the pointers
11       SortedLists = Lists of contiguous arrays with equal keys in currBytes
12       If (sizeof(SortedLists)=0 or currBytes = b-4) sorted = true
13       else currBytes = nextfourBytes
14   end for
```

ROUTINE 4.2: *GPUTeraSort: This routine is used to sort an input sequence of length n and b byte keys. The input sequence is copied into a 2D-texture, whose width and height is set to a power-of-2 that is closest to* $\sqrt{n}$ *(line 2). The sorting algorithm starts using the first four bytes (Line 4). Next, it performs at most* $\frac{b}{4}$ *scans on a subset of the input sequence and during each stage, performing a fast bitonic radix sort on the GPU (line 9). Then it reads back the pointers and computes contiguous portions of the array to be sorted. These contiguous unsorted portions consist of the same key till the value of currBytes and are processed further based on the remaining bytes (Line 13).*

4.3 Handling Wide Keys

GPUTeraSort uses a hybrid radix-bitonic sort algorithm. It uses bitonic sort on the first few bytes of the keys as the initial radix for sorting. In order to handle string comparisons using 32-bit floating point hardware, it is important to encode data on the GPU but avoid special IEEE floating point values such as NaN, ∞, etc. by masking the two most significant bits (MSB) to "10". The key encoding to GPU floating point representations handles any data type, including ASCII strings with lexicographic comparisons. After the GPU orders the key-pointer array based on the first few bytes, GPUTeraSort scans the array to identify contiguous portions of the array with equal keys. It sorts these portions based on the second set of bytes on the GPU. It repeats the process till the entire array is sorted or all the key bytes have been compared. Routine 4.2 gives a pseudo-code for GPUTeraSort's hybrid radix-bitonic sort using the GPU.

We implemented three prior GPU-based sorting algorithms [20, 22, 35] and compared their performance to GPUTeraSort. Figure 6 shows the comparison - GPUTeraSort bitonic sort has a 3x to 6x advantage over previous GPU sorting algorithms.

4.4 Cache-Efficient Sorting on GPUs

In this section, we investigate the cache-performance of the fragment processors while performing each step in the bitonic sorting algorithm. Suppose the width of the input array is W and the height is H, and the block size is $B \times B$. In each step, each pixel is compared against one other pixel. Therefore, there are atleast $n_{compulsory} = \frac{W \times H}{B^2}$ cache misses. Our goal is to reduce the total number of cache

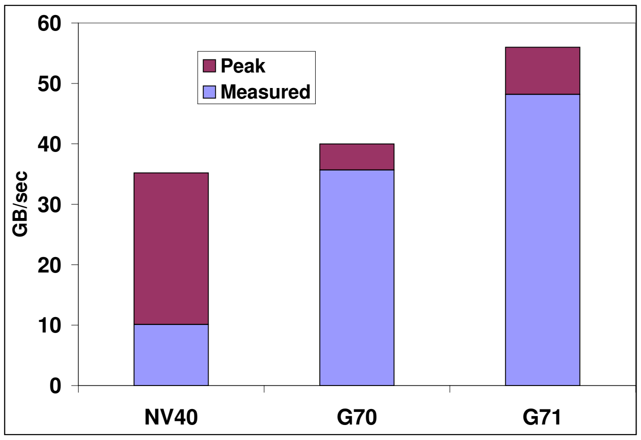

Figure 7: GPUTeraSort performs cache-efficient computations to achieve almost peak memory performance. This graph highlights the peak memory performance and the measured memory bandwidth on three high-end GPUs released in successive generations. Observe that the measured memory bandwidth is over 90% of the peak bandwidth on NVIDIA 7800 (G70) and 7900 GPUs (G71).

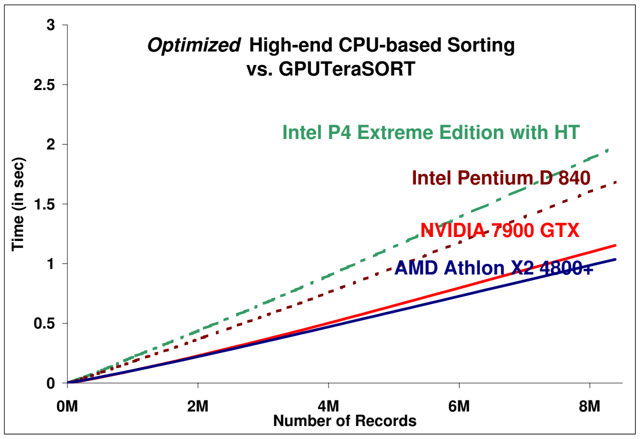

Figure 8: Comparison of GPUTeraSort performance to a hand-optimized CPU-based Quicksort implementation on high-end multi-core CPUs. The CPU-based sorting algorithm has 90% usage for performing key-pointer sorting alone. GPUTeraSort offloads the CPU computations to a GPU and achieves comparable performance to highly-optimized CPU-based sorting algorithms running on expensive CPUs.

misses to a value close to $n_{compulsory}$ in each step.

As described in Section 4.1, each step renders either row-aligned or column-aligned quadrilaterals in order to perform comparisons. Furthermore, the elements in these regions access fixed-distance elements in adjacent row-aligned or column-aligned regions respectively for comparisons. Without loss of generality, assume it is rendering a row-aligned quad of height h and width W. Then, each rendering operation fetches $n_{blocks} = \frac{W \times h}{B^2}$ blocks. If these blocks are not in the cache, they cause cache misses. The cache analysis has two cases based on the height of the row-aligned quad.

Case 1: $h \geq B$. In this case, cache misses in rendering the quad are unavoidable. Therefore, in these steps, we do not modify our algorithm presented in section 4.1.

Case 2: $h < B$. In this case, many cache misses occur if n_{blocks} do not fit in the cache because the cache blocks fetched at the beginning of the quad are mostly evicted by the end of the quad. We use a technique similar to blocking to minimize cache misses. We decompose each quad into multiple quads of width B and height h. We then perform computation on all the quads lying within the $B \times B$ block. The block decomposition effectively reduces the cache misses and achieves close-to-peak memory performance (see Fig.7).

Further details on our caching algorithm are available in [21].

4.5 Analysis

This section presents a theoretical and experimental analysis of GPUTeraSort performance and compares it to other CPU-based and GPU-based sorting algorithms.

4.5.1 Computational Performance

Theoretically, GPUTeraSort has a computational complexity of $O(\frac{nlog^2n}{2})$ and has better performance than periodic balanced sorting network (PBSN) [20]. In practice, on a NVIDIA 7900 GTX GPU GPUTeraSort achieves 14 GOPs/sec processor performance and a memory bandwidth of 49 GB/s. This is close to the peak memory bandwidth (56 GB/s) on the NVIDIA 7900 GTX GPU. We measured the performance on different GPUs, CPUs and compared it against prior sorting algorithms. Figure 6 highlights the performance improvement obtained using GPUTeraSort's hybrid bitonic sorting algorithm against three prior GPU-based algorithms [20, 22, 35] in sorting 32-bit floating point key-pointer pairs on NVIDIA 7800 GTX GPU. The graph indicates at least a 3 times speedup over previous GPU-based algorithms. We also compared the performance of GPUTeraSort with hand-optimized CPU-based algorithms. Figure 8 highlights GPUTeraSort's performance against optimized implementation of QuickSort algorithms on high-end CPUs for varying data sizes. The graph indicates that GPUTeraSort running on an NVIDIA 7900 GTX GPU performs comparably to the optimized CPU-based implementation on an expensive CPU.

4.5.2 Bandwidth Requirements

GPUTeraSort also has relatively low bandwidth requirements. Given n data values, it transfers the data to and from the GPU once. The overall main memory bandwidth requirement of $O(n)$ is significantly lower than the total computational cost i.e., $O(nlog^2n)$. In practice, we observed that the data transfer time is less than 10% of the total sorting time, and indicates that GPUTeraSort is not bandwidth limited (see Figure 9).

4.5.3 Performance and Memory Growth

GPUTeraSort uses the rasterization capabilities of the GPUs to efficiently perform sorting. Therefore, the performance growth rate of GPUTeraSort is directly related to the performance growth rate of GPUs. Fig. 10 indicates the computational time, measured memory bandwidth and performance in sec, GB/sec and GOPs/sec on three successive generation high-end NVIDIA GPUs. The performance

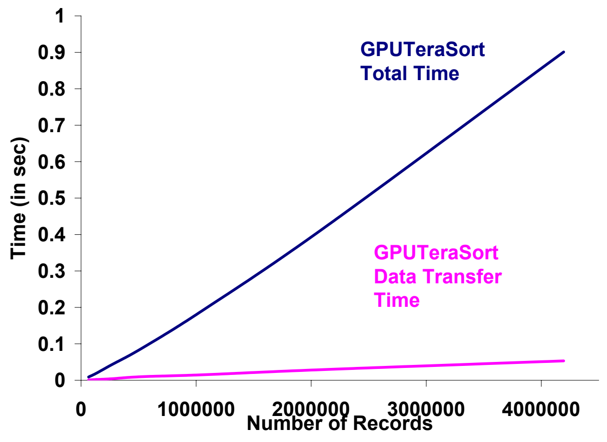

Figure 9: The time taken by GPUTeraSort on an NVIDIA 7800 GTX GPU to sort key-pointers with 32-bit floating point keys including the data transfer overhead. Observe that the algorithm has very low data transfer overhead. In practice, the data transfer cost is less than 10% of the total computational time.

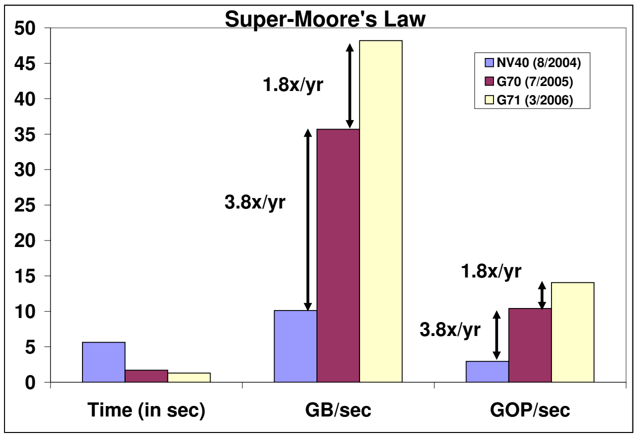

Figure 10: Super-Moore's law performance of GPUTeraSort: This graph shows the memory and performance growth rate of GPUTeraSort in sorting 8M key-pointer pairs on three successive generation high-end GPUs. We observe that the performance of GPUs is improving nearly twice a year.

of GPUTeraSort is improving nearly twice a year.

5. PERFORMANCE

We implemented GPUTeraSort on an inexpensive PC (Table 1) and benchmarked it using the SortBenchmark [7] with the following GPUs: NVIDIA GeForce 6800, NVIDIA GeForce 6800 Ultra, and NVIDIA GeForce 7800 GT. We also tested the performance of optimized CPU-based algorithms on a high-end Dual Xeon server. Specifically, we compared against nsort [31], a commercial CPU-based sorting algorithm with high I/O performance.

5.1 SortBenchmark

The SortBenchmark is designed to measure the IO power and price performance of modern computers [7] based on the sort time as well as the cost of the underlying hardware. The Sort input is a disk file of 100-byte records. The first 10 bytes of each record is a random key value. The sort creates an output file which is a key-ascending permutation of the input file.

The PennySort benchmark measures price/performance. Specifically, the PennySort benchmark measures how many records can be sorted for a penny, assuming the system is depreciated over three years. The PennySort benchmark is further classified into Indy and Daytona categories to measure specially built and general-purpose hardware and software - can the system sort arbitrary files or is it only able to sort files with 10-byte keys in 100-byte records? GPUTeraSort is a general-purpose sorter that qualifies for both categories.

We measured GPUTeraSort's performance on a 3 GHz Pentium IV processor, 2 GB PC5400 main memory, a SuperMicro Marvel MV8 SATA controller connecting 8 Western Digital Caviar SATA disks running Windows XP SP2, and a wide range of commodity GPUs costing less than $300 and used OpenGL to program the GPUs.

Table 1 presents the costs associated with different PC components and the performance obtained using these components in a PC. Observe that a commodity PC with an NVIDIA 7800 GT GPU is able to sort 60GB of data in 648 seconds, this outperforms the current PennySort benchmark in both the Indy (40GB) and Daytona (15GB) categories.

Figure 11 shows the time spent in the different stages of GPUTeraSort Phase 1 on the NVIDIA 6800, 6800 Ultra and 7800 GT GPUs. We configured four SATA disks as the input RAID0 disks and four SATA disks as temporary storage disks and measured the performance on a 100GB data file. Observe that a 7800 GT is able to achieve an almost peak read performance of 220 MB/sec, a write performance of 200 MB/sec and an overall Phase 1 sort performance of 200 MB/s on Phase 1 of a 100MB sort. The 6800 Ultra and 6800 GPUs achieve 185 MB/sec and 147 MB/s respectively during Phase 1. Figure 11 indicates that peak I/O performance can be achieved by co-processing on GPUs. The graph also indicates that time spent in ordering key-pointers (the GPU processing time) is significantly lower than the other stages for the 7800GT GPU. It is possible to achieve higher performance by improving the I/O performance using more disks and a faster main memory.

5.2 Database Sizes

The performance of any external sorting algorithm is largely dependent on its I/O performance, in both Phases 1 and 2. The I/O performance of phase 2 is dependent on the data partition size and layout generated in Phase 1.

We measured the performance of GPUTeraSort and nsort Phase 1 by varying the sort file sizes from 20GB to 100GB 12. Both systems used 4 input/output disks and 4 temporary disks in a RAID0 plus JBOD configuration. We measured the performance of nsort on a 3 GHz Pentium IV and 3.6 GHz Dual Xeon CPUs. We measured GPUTeraSort on the same (slow) 3 GHz Pentium IV with NVIDIA 6800, 6800 Ultra and 7800 GT GPUs. The graph indicates that the 7800 GT is able to sort 100 MB files faster than a Dual Xeon processor. As a result, GPUTeraSort will be able to handle larger databases when it has a powerful GPU.

Figure 13 measures GPUTeraSort's I/O performance in

PC Components	Price (in $)
Super Micro PDSGE motherboard	220
OCZ Corsair PC 5400 RAM	164
Pentium IV CPU	177
9 Western Digital 80 GB disks	477
Power Supply	75
Case	70
SuperMicro SATA Controller	95
System Cost (without GPU)	1,278

GPU Model	6800	6800 Ultra	7800 GT
GPU Cost ($)	120	280	265
Total PC Cost (with GPU)($)	1,398	1,558	1,543
PennySort time limit	715 sec	641 sec	648 sec
GPUTeraSort	56 GB	58 GB	60 GB

Table 1: GPUTeraSort PennySort benchmark record: the pricing of the components used in the PennySort benchmarks. The three system prices give three time budgets and 3 results. The best result is 60 GB using an NVIDIA 7800 GT GPU. GPUTeraSort averages 185 MB/s during each of Phases 1 and 2. This performance is comparable to the performance obtained using dual 3.6 GHz Xeon processors costing $2200.

Phase I on different GPUs and the Phase 1 I/O performance of nsort (labeled "Pentium 4" in the graph) as a function of the number of input and temporary disks. The input disks are configured RAID0 and the temporary disks are configured as just a bunch of disks (JBOD). The figure indicates that nsort-Pentium IV achieves a peak I/O performance of 150 MB/s while nsort on dual Xeons and GPUTeraSort on slow CPUs achieve peak I/O performance. It shows that GPUs can improve I/O performance for external memory sorting. The graph also indicates that GPUTeraSort is able to achieve higher I/O performance on RAID0 configurations with 2 and 3 input disks than high-end dual Xeon processors.

Figure 14 shows the GPUTeraSort elapsed time for various input data sizes. The 7800 GT outperforms the Xeon processors as the size of the database increases, mainly because the 7800 GT GPU is able to efficiently partition the input file into large data chunks in Phase 1, thereby improving the I/O performance in Phase 2.

6. ANALYSIS

GPUTeraSort's performance depends on several factors:

- **Hardware performance:** The performance of GPUTeraSort is dependent on the performance of the underlying hardware. As illustrated in Fig. 11, the overall performance of the algorithm depends not only on I/O performance, but also the performance of the GPU, the CPU, the memory, and the performance of the interconnects among the different system components.
- **Load Balancing:** The sorting algorithm uses task parallelism to perform I/O, memory, and sorting operations efficiently and concurrently. Therefore, the performance of the overall algorithm is based on the pipeline load-balancing and effective scheduling of the pipeline stages and using all the memory bandwidth of.

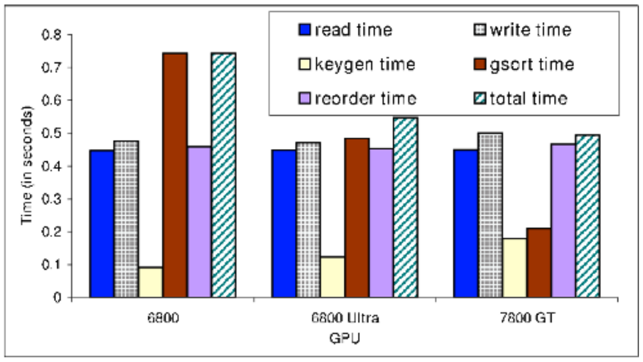

Figure 11: Performance breakdown of the different stages in the phase I of GPUTeraSort algorithm on three GPUs - NVIDIA 6800, NVIDIA 6800 Ultra and NVIDIA 7800 GT. The graphs indicate almost peak I/O performance using an NVIDIA 7800 GT GPU. Observe that I/O is the pipeline bottleneck for the 7800; performance would improve with more disks and faster RAM.

- **Key sizes:** GPUTeraSort is based on a hybrid bitonic-radix sort. In the worst-case, it may need to perform multiple sorting operations on the entire array based on the distribution of the keys. Specifically, it achieves a worst-case performance when all the input keys are equal.
- **Database sizes:** GPUTeraSort's performance depends on the input data size and the time taken to transfer the data to the GPU. In practice, the data transfer time is significantly lower than the computation time with the exception of very small databases.
- **Partition sizes:** GPUTeraSort's performance varies as a function of the run sizes chosen in Phase 1 as is the case with all sorting algorithms.

7. LIMITATIONS

GPUTeraSort has many limitations. These include

- **Variable-sized keys:** GPUTeraSort's is based on radix sort and works well for keys with fixed number of bytes. It works correctly on variable-sized keys.
- **Almost sorted databases:** GPUTeraSort does not benefit if the input data file is almost sorted, while databases, adaptive sorting algorithms work better on such data.
- **Programmable GPUs:** GPUTeraSort requires programmable GPUs with fragment processors. Most desktop and laptop GPUs manufactured after 2003 have these capabilities. However, some low-end GPUs (for PDAs or mobile phones) may not have these capabilities.
- **Memory sizes:** Due to the pipelined design of Phase 1 of GPUTeraSort's run size of Phase 1 is limited to 1/5 of the total main memory size.

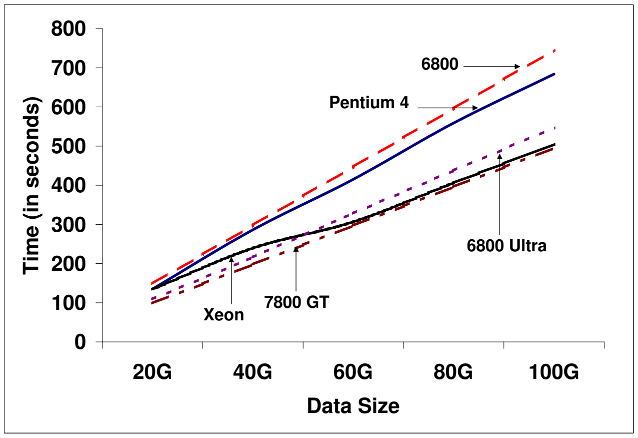

Figure 12: GPUTeraSort and nsort (solid lines) Phase 1 performance vs file size. GPUTeraSort runs are 100MB while nsort runs are less than 25MB in order to fit the key-pointers in the CPU L2 cache. GPUTeraSort on a 7800 GT GPU has performance comparable to an expensive 3.6 GHz Dual Xeon server.

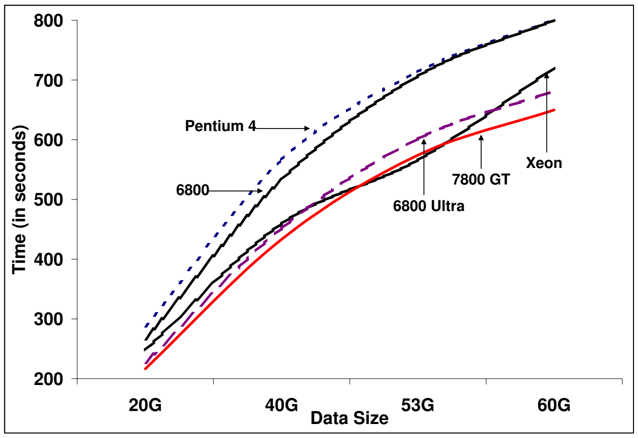

Figure 14: Total sort time for GPUTeraSort and nsort (Pentium 4) shows the performance of GPUTeraSort on different mid-range GPUs and nsort on high-end dual Xeon processors. Observe that GPU-assisted sorts are competitive with CPU-only sorts for larger databases – yet the hardware is much less expensive.

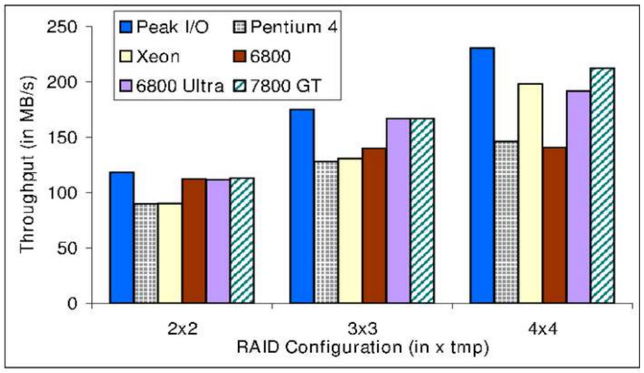

Figure 13: I/O Performance on different input and temporary disk counts: GPUTeraSort using a 7800GT GPU achieves near-peak I/O performance on all disk configurations.

8. CONCLUSIONS AND FUTURE WORK

We presented GPUTeraSort, a novel sorting architecture that quickly sorts billion-record datasets. It uses GPUs to handle wide keys, long records, many data types, and input sizes. It handles databases that do not fit in the GPU video memory or in the main memory.

We evaluated GPUTeraSort on various benchmarks and compared its performance with optimized CPU-based algorithms. The results indicate that graphics co-processors can significantly improve the I/O performance and scale well on massive databases. The overall performance of GPUTeraSort with a mid-range GPU (costing around $300) is comparable to that of a CPU-based algorithm running on a high-end dual Xeon processors (costing around $2,200). In practice, GPUTeraSort achieves a good price-performance and outperforms the current PennySort benchmarks.

There are several avenues for future work. We are interested in extending GPUTeraSort to clusters of PCs with multiple GPUs. Another interesting avenue is to accelerate the performance of Phase 2 of the algorithm using GPUs. It is worth considering integrating GPUTeraSort with SQL systems and data streaming algorithms.

Current processors and disks have high power requirements and generate considerable amount of heat. High temperatures can affect the stability of the systems and require effective cooling solutions. Furthermore, power can be a limiting factor to the system performance and can be expensive over time. An interesting avenue is to design power- and temperature-efficient algorithms using GPUs.

Acknowledgements

This work is supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, DARPA/RDECOM Contract N61339-04-C-0043, ONR Contract N00014-01-1-0496 and Intel. Thanks to Chris Nyberg for his help in understanding how to configure nsort and much advice on IO configuration for sorting. We would like to thank Craig Peeper, Peter-Pike Sloan, Jingren Zhou, and David Blythe for their useful feedback. Thanks to David Tuft and other members of UNC GAMMA group for their useful suggestions and support.

9. REFERENCES

[1] Clearspeed technology. *(http://www.clearspeed.com)*, 2005.
[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] Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
[4] A. Ailamaki. Database architectures for new hardware. *VLDB Tutorial Notes*, 2004.
[5] 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.

[6] 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.

[7] Anon. et al. A measure of transaction processing power. *Datamation*, page 112, 1 1985.

[8] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):243–254, June 1997.

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

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

[11] B. Baugsto, J. Griepsland, and J. Kamerbeck. Sorting large data files on poma. *Proc. of COMPAR-90 VAPPIV*, pages 536–547, 1990.

[12] 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.

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

[14] D. Dewitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. *Proc. of International Conference on Parallel and Distributed Information Systems*, 1991.

[15] Anonymous et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985.

[16] E. Friend. Sorting on electronic computer systems. *Journal of the ACM*, 3(3):134–168, 1956.

[17] P. Garcia and H. F. Korth. Multithreaded architectures and the sort benchmark. *Proc. of First International Workshop on the Data Management on New Hardware*, 2005.

[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 and D. Manocha. Efficient relational database management on graphics processors. *Proc. of ACM Workshop on Data Management on New Hardware*, 2005.

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

[21] Naga K. Govindaraju, Jim Gray, and Dinesh Manocha. Cache-efficient general purpose algorithms on GPUs. Technical report, Microsoft Research, December 2005.

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

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

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

[25] X. Li, G. Linoff, S. Smith, C. Stanfill, and K. Thearling. A practical external ort for shared disk mpps. *Proc. of SuperComputing*, pages 666–675, 1993.

[26] 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.

[27] 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.

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

[29] 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.

[30] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[31] Nsort: Fast parallel sorting. http://www.ordinal.com/.

[32] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A risc machine sort. *SIGMOD*, pages 233–242, 1994.

[33] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A cache-sensitive parallel external sort. *VLDB Journal: Very Large Data Bases*, 4(4):603–627, 1995.

[34] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. 2005.

[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] Betty Salzberg, Alex Tsukerman, Jim Gray, Michael Stuewart, Susan Uren, and Bonnie Vaughan. FastSort: A distributed single-input single-output external sort. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):94–101, June 1990.

[39] 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.

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

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

[42] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, pages 209–271, 2001.

[43] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12:148–169, 1994.

[44] M. Zagha and G. Blelloch. Radix sort for vector multiprocessors. *Proc. of SuperComputing*, 1991.

[45] 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.