Homayoun

Reference 17

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 $\sim$ 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:

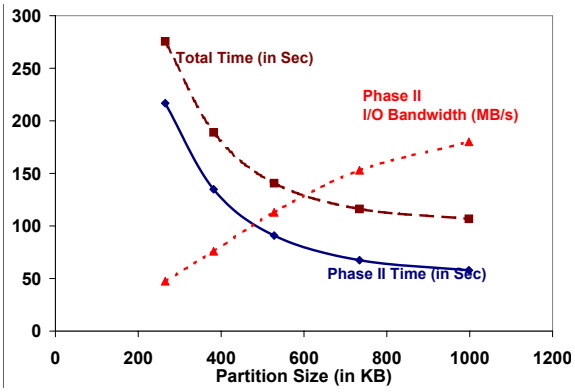

Figure 1: Performance of an optimized merge-based external memory sorting algorithm on a Dual 3.6 GHz Xeon processor system. Observe that the speed of Phase 2 increases nearly linearly with the partition size. As the data partition sizes in Phase I fit well in the L2 cache sizes, the Phase 1 time remains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

- **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

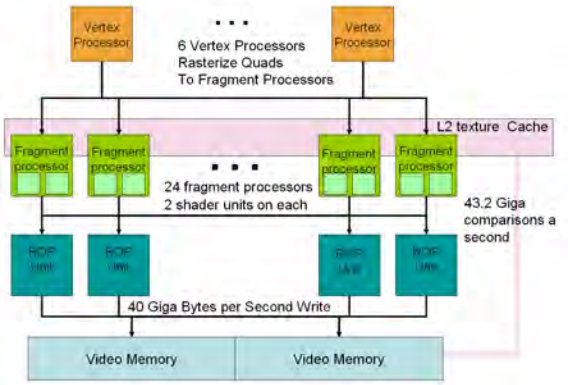

Figure 2: This figure highlights the high data parallelism and memory bandwidth inside a GPU. GPUTeraSort uses the vector processing functionalities to implement a highly parallel bitonic sorting network. It outperforms prior CPU-based and GPU-based algorithms by 3-10 times.

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.