Designing Efficient Sorting Algorithms for Manycore GPUs

Nadathur Satish
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA
Email: nrsatish@eecs.berkeley.edu

Mark Harris Michael Garland
NVIDIA Corporation
Santa Clara, CA
Email: mharris@nvidia.com, mgarland@nvidia.com

***Abstract*—We describe the design of high-performance parallel radix sort and merge sort routines for manycore GPUs, taking advantage of the full programmability offered by CUDA. Our radix sort is the fastest GPU sort and our merge sort is the fastest comparison-based sort reported in the literature. Our radix sort is up to 4 times faster than the graphics-based GPUSort and greater than 2 times faster than other CUDA-based radix sorts. It is also 23% faster, on average, than even a very carefully optimized multicore CPU sorting routine.**

To achieve this performance, we carefully design our algorithms to expose substantial fine-grained parallelism and decompose the computation into independent tasks that perform minimal global communication. We exploit the high-speed on-chip shared memory provided by NVIDIA's GPU architecture and efficient data-parallel primitives, particularly parallel scan. While targeted at GPUs, these algorithms should also be well-suited for other manycore processors.

I. Introduction

Sorting is a computational building block of fundamental importance and is one of the most widely studied algorithmic problems [1], [2]. The importance of sorting has also lead to the design of efficient sorting algorithms for a variety of parallel architectures [3]. Many algorithms rely on the availability of efficient sorting routines as a basis for their own efficiency, and many other algorithms can be conveniently phrased in terms of sorting. Database systems make extensive use of sorting operations [4]. The construction of spatial data structures that are essential in computer graphics and geographic information systems is fundamentally a sorting process [5], [6]. Efficient sort routines are also a useful building block in implementing algorithms like sparse matrix multiplication and parallel programming patterns like MapReduce [7], [8]. It is therefore important to provide efficient sorting routines on practically any programming platform, and as computer architectures evolve there is a continuing need to explore efficient sorting techniques on emerging architectures.

One of the dominant trends in microprocessor architecture in recent years has been continually increasing chip-level parallelism. Multicore CPUs—providing 2–4 scalar cores, typically augmented with vector units—are now commonplace and there is every indication that the trend towards increasing parallelism will continue on towards "manycore" chips that provide far higher degrees of parallelism. GPUs have been at the leading edge of this drive towards increased chip-level parallelism for some time and are already fundamentally manycore processors. Current NVIDIA GPUs, for example, contain up to 240 scalar processing elements per chip [9], and in contrast to earlier generations of GPUs, they can be programmed directly in C using CUDA [10], [11].

In this paper, we describe the design of efficient sorting algorithms for such manycore GPUs using CUDA. The programming flexibility provided by CUDA and the current generation of GPUs allows us to consider a much broader range of algorithmic choices than were convenient on past generations of GPUs. We specifically focus on two classes of sorting algorithms: a *radix sort* that directly manipulates the binary representation of keys and a *merge sort* that requires only a comparison function on keys.

The GPU is a massively multithreaded processor which can support, and indeed expects, several thousand concurrent threads. Exposing large amounts of fine-grained parallelism is critical for efficient algorithm design on such architectures. In radix sort, we exploit the inherent fine-grained parallelism of the algorithm by building our algorithm upon efficient parallel scan operations [12]. We expose fine-grained parallelism in merge sort by developing an algorithm for pairwise parallel merging of sorted sequences, adapting schemes based on parallel splitting and binary search previously described in the literature [13], [14], [15]. We demonstrate how to impose a block-wise structure on the sorting algorithms, allowing us to exploit the fast on-chip memory provided by the GPU architecture. We also use this on-chip memory for locally ordering data to improve coherence, thus resulting in substantially better bandwidth utilization for the scatter steps used by radix sort.

Our experimental results demonstrate that our radix sort algorithm is faster than all previously published GPU sorting techniques when running on current-generation NVIDIA GPUs. Our tests further demonstrate that our merge sort algorithm is the fastest comparison-based GPU sort algorithm described in the literature, and is faster in several cases than other GPU-based radix sort implementations. Finally, we demonstrate that our radix sort is highly competitive with multicore CPU implementations, being up to 4.1 times faster than comparable routines on an 8-core 2.33 GHz Intel E5345 system and on average 23% faster than the most

To Appear in Proc. 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.

optimized published multicore CPU sort [16] running on a 4-core 3.22 GHz Intel Q9550 processor.

II. Parallel Computing on the GPU

Before discussing the design of our sorting algorithms, we briefly review the salient details of NVIDIA's current GPU architecture and the CUDA parallel programming model. As their name implies, GPUs (Graphics Processing Units) came about as accelerators for graphics applications, predominantly those using the OpenGL and DirectX programming interfaces. Due to the tremendous parallelism inherent in graphics, GPUs have long been massively parallel machines. Although they were originally purely fixed-function devices, GPUs have rapidly evolved into increasingly flexible programmable processors.

Modern NVIDIA GPUs—beginning with the GeForce 8800 GTX—are fully programmable manycore chips built around an array of parallel processors [9], as illustrated in Figure 1. The GPU consists of an array of SM multiprocessors, each of which is capable of supporting up to 1024 co-resident concurrent threads. NVIDIA's current products range in size from 1 SM at the low end to 30 SMs at the high end. A single SM shown in Figure 1 contains 8 scalar SP processors, each with 1024 32-bit registers, for a total of 64KB of register space per SM. Each SM is also equipped with a 16KB on-chip memory that has very low access latency and high bandwidth, similar to an L1 cache.

All thread management, including creation, scheduling, and barrier synchronization is performed entirely in hardware by the SM with essentially zero overhead. To efficiently manage its large thread population, the SM employs a SIMT (Single Instruction, Multiple Thread) architecture [9], [10]. Threads are executed in groups of 32 called *warps*. The threads of a warp are executed on separate scalar processors which share a single multithreaded instruction unit. The SM transparently manages any divergence in the execution of threads in a warp. This SIMT architecture allows the hardware to achieve substantial efficiencies while executing non-divergent data-parallel codes.

CUDA [10], [11] provides the means for developers to execute parallel programs on the GPU. In the CUDA programming model, an application is organized into a sequential *host* program that may execute parallel programs, referred to as *kernels*, on a parallel *device*. Typically, the host program executes on the CPU and the parallel kernels execute on the GPU, although CUDA kernels may also be compiled for efficient execution on multicore CPUs [17].

A kernel is a SPMD-style (Single Program, Multiple Data) computation, executing a scalar sequential program across a set of parallel threads. The programmer organizes these threads into *thread blocks*; a kernel thus consists of a grid of one or more blocks. A thread block is a group of concurrent threads that can cooperate amongst themselves through barrier synchronization and a per-block shared memory space private to that block. When invoking a kernel, the programmer specifies both the number of blocks and the number of threads per block to be created when launching the kernel. The thread blocks of a CUDA kernel essentially virtualize the SM multiprocessors of the physical GPU. We can think of each thread block as a virtual multiprocessor where each thread has a fixed register footprint and each block has a fixed allocation of per-block shared memory.

A. Efficiency Considerations

The SIMT execution of threads is largely transparent to the CUDA programmer. However, much like cache line sizes on traditional CPUs, it can be ignored when designing for correctness, but must be carefully considered when designing for peak performance. To achieve best efficiency, kernels should avoid execution divergence, where threads within a warp follow different execution paths. Divergence between warps, however, introduces no performance penalty.

The on-chip shared memory provided by the SM is an essential ingredient for efficient cooperation and communication amongst threads in a block. It is particularly advantageous when a thread block can load a block of data into on-chip shared memory, process it there, and then write the final result back out to external memory.

The threads of a warp are free to load from and store to any valid address, thus supporting general gather and scatter access to memory. However, when threads of a warp access consecutive words in memory, the hardware is able to coalesce these accesses into aggregate transactions with the memory system, resulting in substantially higher memory throughput. For instance, a warp of 32 threads gathering from widely separated addresses will issue 32 requests to memory, while a warp reading 32 consecutive words will issue 2 requests.

Finally, the GPU relies on multithreading, as opposed to a cache, to hide the latency of transactions with external memory. It is therefore necessary to design algorithms that create enough parallel work to keep the machine fully utilized. For current-generation hardware, a minimum of around 5,000–10,000 threads must be live simultaneously to efficiently utilize the entire chip.

B. Algorithm Design

When designing algorithms in the CUDA programming model, our central concern is in dividing the required work up into pieces that may be processed by p thread blocks of t threads each. Throughout this paper, we will use a thread block size of $t = 256$. Using a power-of-2 size makes certain algorithms, such as parallel scan and bitonic sort, easier to implement, and we have empirically determined that 256-thread blocks provide the overall best performance. For operations such as sort that are required to process input arrays of size n, we choose $p \propto n/t$, assigning a single or small constant number of input elements to each thread.

We treat the number of thread blocks as the analog of "processor count" in parallel algorithm analysis, despite the fact that threads within a block will execute on multiple processing cores within the SM multiprocessor. It is at the thread block (or SM) level at which internal communication

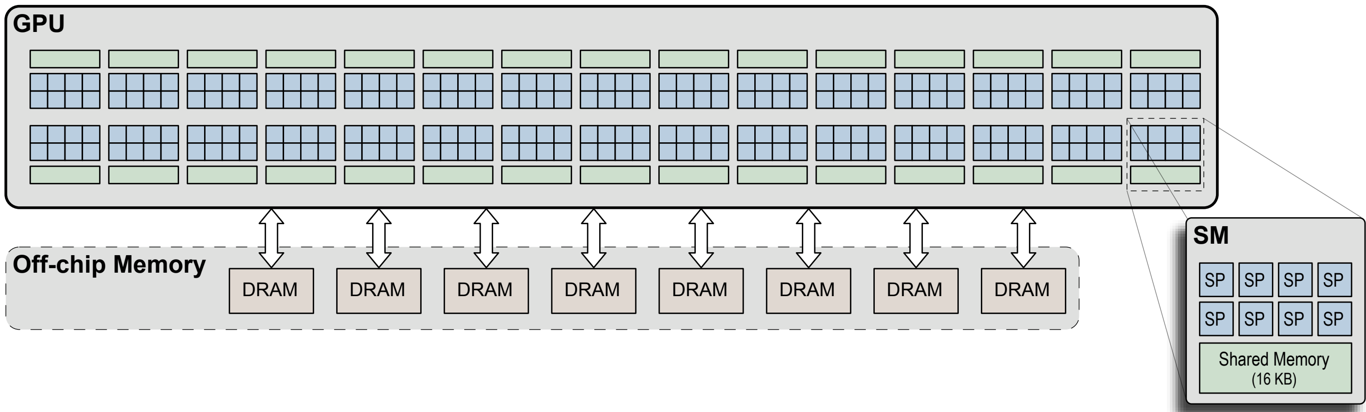


Fig. 1. GeForce GTX 280 GPU with 240 scalar processor cores (SPs), organized in 30 multiprocessors (SMs).

is cheap and external communication becomes expensive. Therefore, focusing on the decomposition of work between the p thread blocks is the most important factor in assessing performance.

It is fairly natural to think of a single thread block as the rough equivalent of a CRCW asynchronous PRAM [18], using explicit barriers to synchronize. Broadly speaking, we find that efficient PRAM algorithms are generally the most efficient algorithms at the block level. Since global synchronization can only be achieved via the barrier implicit between successive kernel calls, the need for global synchronization drives the decomposition of parallel algorithms into separate kernels.

III. Radix Sort

Radix sort is one of the oldest and best-known sorting algorithms, and on sequential machines it is often amongst the most efficient for sorting small keys. It assumes that the keys are d-digit numbers and sorts on one digit of the keys at a time, from least to most significant. For a fixed key size d, the complexity of sorting n input records will be $O(n)$.

The sorting algorithm used within each of the d passes of radix sort is typically a counting sort or bucket sort [2]. Each radix-2^b digit is a b-bit string within the key. For a given digit of each key, we compute the number of keys whose digits are smaller plus the number of keys with the same digit occurring earlier in the sequence. This will be the output index at which the element should be written, which we will refer to as the *rank* of the element. Having computed the rank of each element, we complete the pass by scattering the elements into the output array. Since this counting is stable (i.e., it preserves the relative ordering of keys with equal digits) sorting each digit from least to most significant is guaranteed to leave the sequence correctly sorted after all d passes are complete.

Radix sort is fairly easy to parallelize, as the counting sort used for each pass can be reduced to a parallel prefix sum, or *scan*, operation [12], [19], [20], [21]. Scans are a fundamental data-parallel primitive with many uses and which can be implemented efficiently on manycore processors like the GPU [22]. Past experience suggests that radix sort is amongst the easiest of parallel sorts to implement [23] and is as efficient as more sophisticated algorithms, such as sample sort, when n/p is small [23], [21].

The simplest scan-based technique is to sort keys 1 bit at a time—referred to by Blelloch [12] as a "split" operation. This approach has been implemented in CUDA by Harris *et al.* [24] and is publicly available as part of the CUDA Data-Parallel Primitives (CUDPP) library [25]. While conceptually quite simple, this approach is not particularly efficient when the arrays are in external DRAM. For 32-bit keys, it will perform 32 scatter operations that reorder the entire sequence. Transferring data to/from external memory is relatively expensive on modern processors, so we would prefer to avoid this level of data movement if possible. One natural way to reduce the number of scatters is to consider more than $b = 1$ bits per pass. To do this efficiently, we can have each of the p blocks compute a histogram counting the occurrences of the 2^b possible digits in its assigned tile of data, which can then be combined using scan [19], [21]. Le Grand [26] and He *et al.* [27] have implemented similar schemes in CUDA. While more efficient, we have found that this scheme also makes inefficient use of external memory bandwidth. The higher radix requires fewer scatters to global memory. However, it still performs scatters where consecutive elements in the sequence may be written to very different locations in memory. This sacrifices the bandwidth improvement available due to coalesced writes, which in practice can be as high as a factor of 10.

To design an efficient radix sort, we begin by dividing the sequence into tiles that will be assigned to p thread blocks. We focus specifically on making efficient use of memory bandwidth by (1) minimizing the number of scatters to global memory and (2) maximizing the coherence of scatters. Data blocking and a digit size $b > 1$ accomplishes the first goal. We accomplish the second goal by using on-chip shared memory to locally sort data blocks by the current radix-2^b digit. This converts scattered writes to external memory into scattered writes to on-chip memory, which is roughly 2 orders of

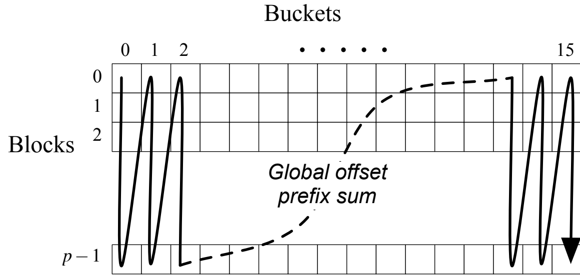


Fig. 2. Per-block histograms to be stored in column-major order for prefix sum.

magnitude faster.

We implement each pass of the radix sort in four phases. As global synchronization is required between each phase, they correspond to separate parallel kernel invocations.

1) Each block loads and sorts its tile in on-chip memory using b iterations of 1-bit split.
2) Each block writes its 2^b-entry digit histogram and the sorted data tile to global memory.
3) Perform a prefix sum over the $p \times 2^b$ histogram table, stored in column-major order, to compute global digit offsets [19], [21] (see Figure 2).
4) Using prefix sum results, each block copies its elements to their correct output position.

As we discussed earlier, our CUDA kernels are executed by blocks of $t = 256$ threads each. While assigning one element per thread would be a natural design choice, handling a larger number of elements per thread is actually somewhat more efficient. We process 4 elements per thread or 1024 elements per block. Performing more independent serial work in each thread improves overall parallel work efficiency and provides more opportunities to hide latency. Since each block will process a tile of 1024 elements, we use $p = \lceil n/1024 \rceil$ blocks in our computations.

The choice of b is determined by two competing factors. We pre-sort each tile in Step 1 to limit the scatter in Step 4 to only 2^b contiguous blocks. Larger values of b will decrease the coherence of this scatter. On the other hand, a small b leads to a large number of passes, each of which performs a scatter in global memory. Given that each block processes $O(t)$ elements, we expect that the number of buckets 2^b should be at most $O(\sqrt{t})$, since this is the largest size for which we can expect uniform random keys to (roughly) fill all buckets uniformly. We find empirically that choosing $b = 4$, which in fact happens to produce exactly $\sqrt{t}$ buckets, provides the best balance between these factors and the best overall performance.

IV. Merge Sort

Since direct manipulation of keys as in radix sort is not always allowed, it is important to provide efficient comparison-based sorting algorithms as well. From the many available choices, we have chosen to explore divide-and-conquer merge sort. As we will see, this leads to an efficient sort and also provides an efficient merge procedure which is a valuable computational primitive in its own right.

Sorting networks, such as Batcher's bitonic sorting network [28], are among the oldest parallel sorting techniques. Bitonic sort is often the fastest sort for small sequences but its performance suffers for larger inputs [23], [21]. This reflects both its asymptotically inefficient $O(n \log^2 n)$ complexity and its relatively high communication cost.

Parallel versions of quicksort can be implemented in parallel using segmented scan primitives [12]. Similar strategies can also be used for other partitioning sorts, such as MSB radix sort which recursively partitions the sequence based on the high order bits of each key [27]. However, the relatively high overhead of the segmented scan procedures leads to a sort that is not competitive with other alternatives [23], [22]. Cederman and Tsigas [29] achieve a substantially more efficient quicksort by using explicit partitioning for large sequences coupled with bitonic sort for small sequences.

Another elegant parallelization technique is used by sample sort [30], [31], [32]. It randomly selects a subset of the input, called *splitters*, which are then sorted by some other efficient procedure. The sorted sequence of splitters can be used to divide all input records into buckets corresponding to the intervals delimited by successive splitters. Each bucket can then be sorted in parallel and the sorted output is simply the concatenation of the sorted buckets. Sample sort has proven in the past to be one of the most efficient parallel sorts on large inputs, particularly when the cost of interprocessor communication is high. However, it appears less effective when n/p, the number of elements per processor, is small [23], [21]. Since we use a fairly small $n/p = 256$, this makes sample sort less attractive. We are also working in a programming model where each of the p thread blocks is given a statically-allocated fixed footprint in on-chip memory. This makes the randomly varying bucket size produced by sample sort inconvenient.

Merge sort is generally regarded as the preferred technique for external sorting, where the sequence being sorted is stored in a large external memory and the processor only has direct access to a much smaller memory. In some respects, this fits the situation we face. Every thread block running on the GPU has access to shared external DRAM, up to 4 GB in current products, but only at most 16 KB of on-chip memory. The latency of working in on-chip memory is roughly 100 times faster than in external DRAM. To keep operations in on-chip memory as much as possible, we must be able to divide up the total work to be done into fairly small tiles of bounded size. Given this, it is also important to recognize that sorting any reasonably large sequence will require moving data in and out of the chip, as the on-chip memory is simply too small to store all data being sorted. Since the cost of interprocessor communication is no higher than the cost of storing data in global memory, the implicit load balancing afforded by merge sort appears more beneficial than sample sort's avoidance of interprocessor communication.

The merge sort procedure consists of three steps:

1) Divide the input into p equal-sized tiles.
2) Sort all p tiles in parallel with p thread blocks.
3) Merge all p sorted tiles.

Since we assume the input sequence is a contiguous array of records, the division performed in Step (1) is trivial. For sorting individual data tiles in Step (2), we use an implementation of Batcher's odd-even merge sort [28]. For sorting t values on chip with a t-thread block, we have found the Batcher sorting networks to be substantially faster than either radix sort or quicksort. We use the odd-even merge sort, rather than the more common bitonic sort, because our experiments show that it is roughly 5–10% faster in practice. All the real work of merge sort occurs in the merging process of Step (3), which we accomplish with a pair-wise merge tree of $\log p$ depth.

At each level of the merge tree, we merge pairs of corresponding odd and even subsequences. This is obviously an inherently parallel process. However, the number of pairs to be merged decreases geometrically. This coarse-grained parallelism is insufficient to fully utilize massively parallel architectures. Our primary focus, therefore, is on designing a process for pairwise merging that will expose substantial fine-grained parallelism.

A. Parallel Merging

Following prior work on merging pairs of sorted sequences in parallel for the PRAM model [13], [33], [14], [15] we use two key insights in designing an efficient merge algorithm. First, we can use a divide-and-conquer technique to expose higher levels of parallelism at each level of the merge tree. And second, computing the final position of elements in the merged sequence can be done efficiently using parallel binary searches.

Given the sorted sequences A and B, we want to compute the sorted sequence $C = \mathrm{merge}(A, B)$. If these sequences are sufficiently small, say of size no greater than $t = 256$ elements each, we can merge them using a single t-thread block. For an element $a_i \in A$, we need only compute $\mathrm{rank}(a_i, C)$, which is the position of element a_i in the merged sequence C. Because both A and B are sorted, we know that $\mathrm{rank}(a_i, C) = i + \mathrm{rank}(a_i, B)$, where $\mathrm{rank}(a_i, B)$ is simply the count of elements $b_j \in B$ with $b_j < a_i$ and which we compute efficiently using binary search. Elements of B can obviously be treated in the same way. Therefore, we can efficiently merge these two sequences by having each thread of the block compute the output rank of its corresponding elements in A and B, subsequently writing those elements to the correct position. Since this can be done in on-chip memory, it will be very efficient.

We merge larger arrays by dividing them up into tiles of size at most t that can be merged independently using the block-wise process we have just outlined. To do so in parallel, we begin by constructing two sequences of *splitters* S_A and S_B by selecting every t-th element of A and B, respectively. By construction, these splitters partition A and B into contiguous tiles of at most t elements each. We now construct a single merged splitter set $S = \mathrm{merge}(S_A, S_B)$, which we achieve with a nested invocation of our merge procedure.

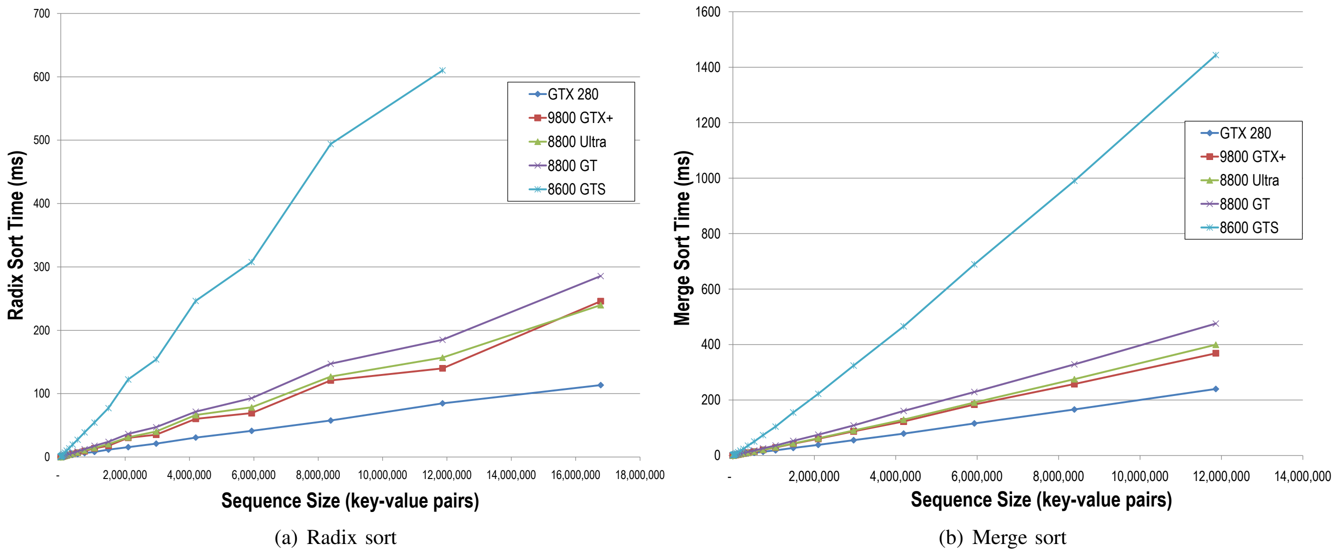
We use the combined set S to split both A and B into contiguous tiles of at most $t = 256$ elements. To do so, we must compute the rank of each splitter s in both input sequences. The rank of a splitter $s = a_i$ drawn from A is obviously i, the position from which it was drawn. To compute its rank in B, we first compute its rank in S_B. We can compute this directly as $\mathrm{rank}(s, S_B) = \mathrm{rank}(s, S) - \mathrm{rank}(s, S_A)$, since both terms on the right hand side are its known positions in the arrays S and S_A. Given the rank of s in S_B, we have now established a t-element window in B in which s would fall, bounded by the splitters in S_B that bracket s. We determine the rank of s in B via binary search within this window.

V. Performance Analysis

We now examine the experimental performance of our sorting algorithms. Our performance tests are all based on sorting sequences of key-value pairs where both keys and values are 32-bit words. Although this does not address all sorting problems, it covers many cases that are of primary importance to us, such as sorting points in space and building irregular data structures. We use a uniform random number generator to produce random keys. We report GPU times as execution time only and do not include the cost of transferring input data from the host CPU memory across the PCI-e bus to the GPU's on-board memory. Sorting is frequently most important as one building block of a larger-scale computation. In such cases, the data to be sorted is being generated by a kernel on the GPU and the resulting sorted array will be consumed by a kernel on the GPU.

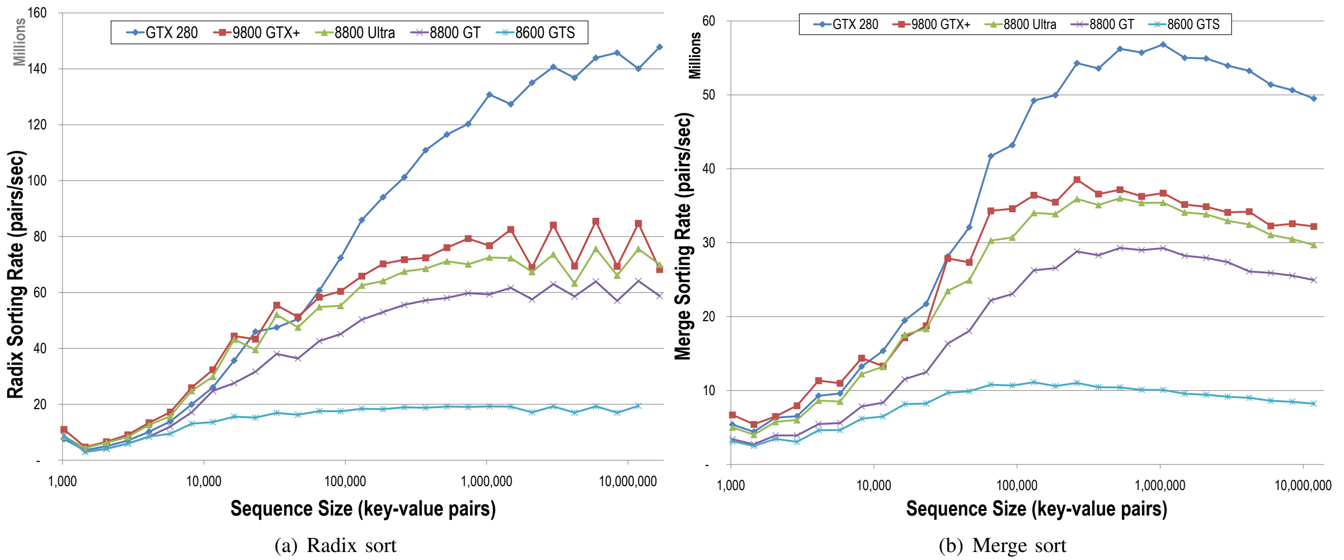
Figure 3 reports the running time for our sorting implementations. The input arrays are randomly generated sequences whose lengths range from 1K elements to 16M elements, only some of which are power-of-2 sizes. One central goal of our algorithm design is to scale across a significant range of physical parallelism. To illustrate this scaling, we measure performance on a range of NVIDIA GeForce GPUs: the GTX 280 (30 SMs), 9800 GTX+ (16 SMs), 8800 Ultra (16 SMs), 8800 GT (14 SMs), and 8600 GTS (4 SMs). Our measurements demonstrate the scaling we expect to see: the progressively more parallel devices achieve progressively faster running times.

The sorting performance trends are more clearly apparent in Figure 4, which shows *sorting rates* derived from Figure 3. We compute sorting rates by dividing the input size by total running time, and thus measure the number of key-value pairs sorted per second. We expect that radix sort, which does $O(n)$ work, should converge to a roughly constant sorting rate, whereas merge sort should converge to a rate of $O(1/\log n)$. For small array sizes, running times can be dominated by fixed overhead and there may not be enough work to fully utilize the available hardware parallelism, but these trends are apparent at larger array sizes. The radix sorting rate exhibits a larger degree of variability, which we believe is due to the load placed on the memory subsystem by radix sort's global scatters.



(a) Radix sort (b) Merge sort

Fig. 3. Sorting time for varying sequence sizes across a range of GPU chips.



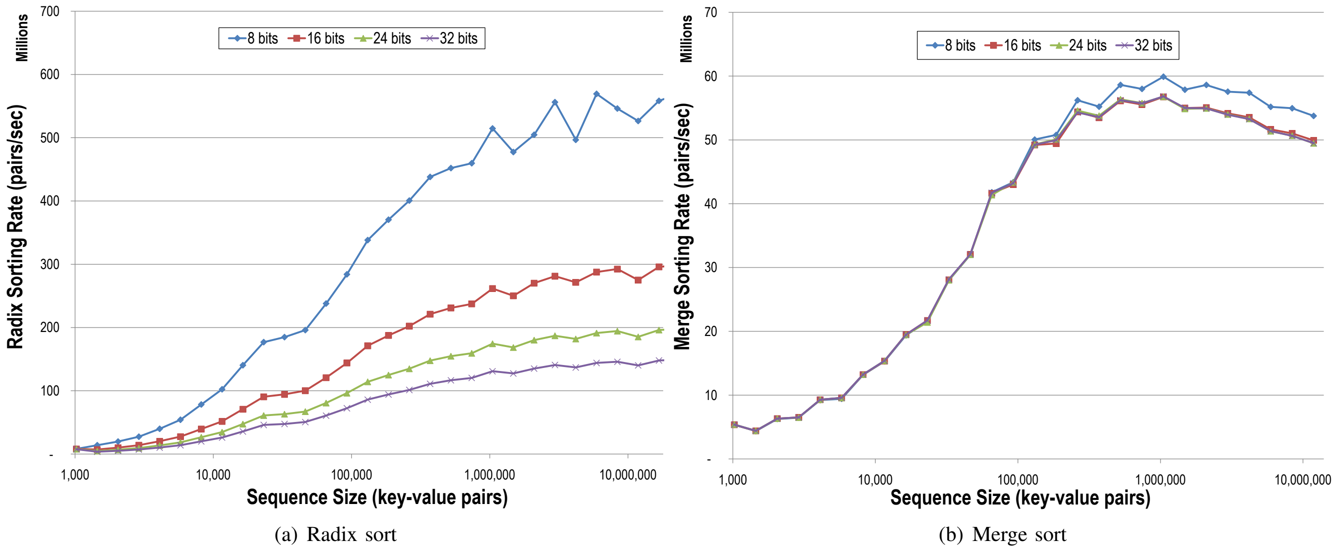
(a) Radix sort (b) Merge sort

Fig. 4. Sorting rates (elements sorted per second) derived from running times shown in Figure 3.

We next examine the impact of different key distributions on our sort performance. Figure 5 shows sorting rates for different key sizes. Here, we have simply restricted the number of random bits we generate for keys to the lower order 8, 16, 24, or full 32 bits. As expected, this makes a substantial difference for radix sort, where we can limit the number of passes over the data we make. Measured sorting rates for 8, 16, and 24-bit keys are, precisely as expected, 4, 2, and 1.33 times faster than 32-bit keys for all but the smallest sequences. For merge sort, the effect is far less dramatic, although having only 256 unique keys (the 8-bit case) does result in 5–10% better performance.

We also adopt the technique of Thearling and Smith [20] for generating key sequences with different entropy levels by computing keys from the bitwise AND of multiple random samples. As they did, we show performance—in Figure 6—using 1–5 samples per key, effectively producing 32, 25.95, 17.41, 10.78, and 6.42 unique bits per key, respectively. We also show the performance for 0 unique bits, corresponding to AND'ing infinitely many samples together, thus making all keys zero. We see that increasing entropy, or decreasing unique bits per key, results in progressively higher performance. This agrees with the results reported by Thearling and Smith [20] and Dusseau *et al.* [21].

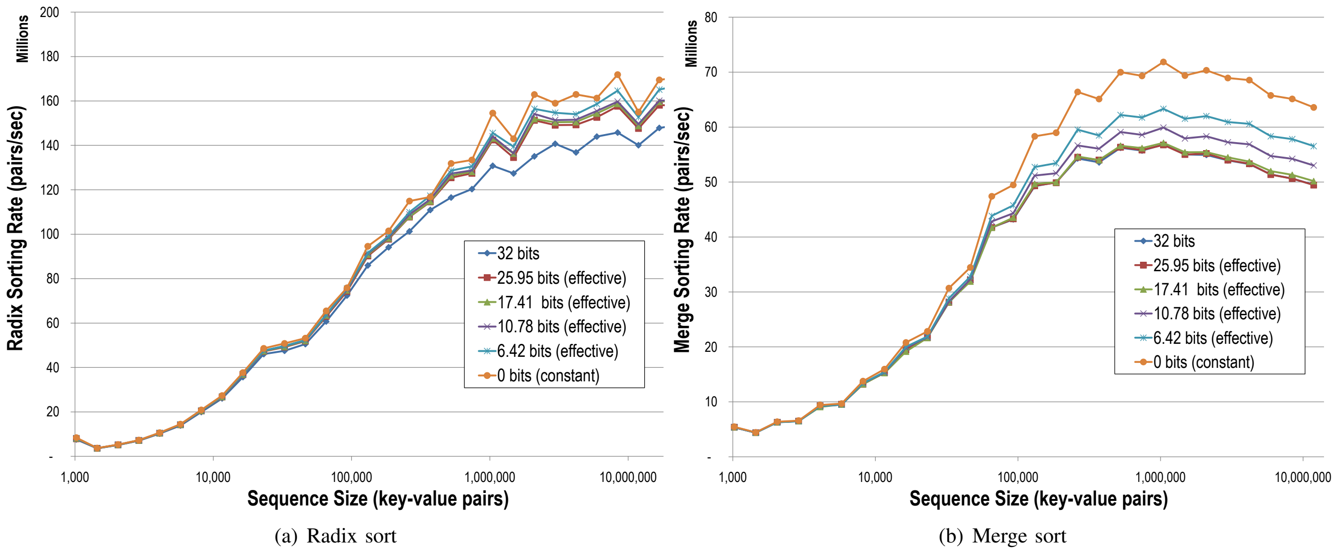
Finally, we also measured the performance of both sorts when the randomly generated sequences were pre-sorted. As expected, this made no measureable difference to the radix sort. Applying merge sort to pre-sorted sequences produced nearly identical measurements to the case of uniform 0 keys shown in Figure 6, resulting in 1.3 times higher performance



(a) Radix sort

(b) Merge sort

Fig. 5. Sort performance on GeForce GTX 280 with restricted key sizes.



(a) Radix sort

(b) Merge sort

Fig. 6. Sort performance on GeForce GTX 280 with varying key entropy.

for sequences of 1M keys or more.

A. Comparison with GPU-based Methods

Figure 7 shows the relative performance of both our sorting routines, as well as the radix sort published by Le Grand [26] in *GPU Gems 3*, the radix sort algorithm implemented by Sengupta *et al.* [22] in CUDPP [25], and the bitonic sort system GPUSort of Govindaraju *et al.* [34]. All performance measurements reflect the running time of the original implementations provided by the respective authors. We measure performance on the 8800 Ultra, rather than the more recent GTX 280, to provide a more fair comparison as some of these older codes have not been tuned for the newer hardware.

GPUSort is an example of traditional graphics-based GPGPU programming techniques; all computation is done in pixel shaders via the OpenGL API. Note that GPUSort only handles power-of-2 input sizes on the GPU, performing post-processing on the CPU for arrays of other sizes. Therefore, we only measure GPUSort performance on power-of-2 sized sequences, since only these reflect actual GPU performance. GPU-ABiSort [35]—another well-known graphics-based GPU sort routine—does not run correctly on current generation GPUs. However, it was previously measured to be about 5% faster than GPUSort on a GeForce 7800 system. Therefore, we believe that the GPUSort performance on the GeForce 8800 should be representative of the GPU-ABiSort performance as well. All other sorts shown are implemented in CUDA.

Several trends are apparent in this graph. First of all, the CUDA-based sorts are generally substantially faster than GPUSort. This is in part due to the intrinsic advantages of

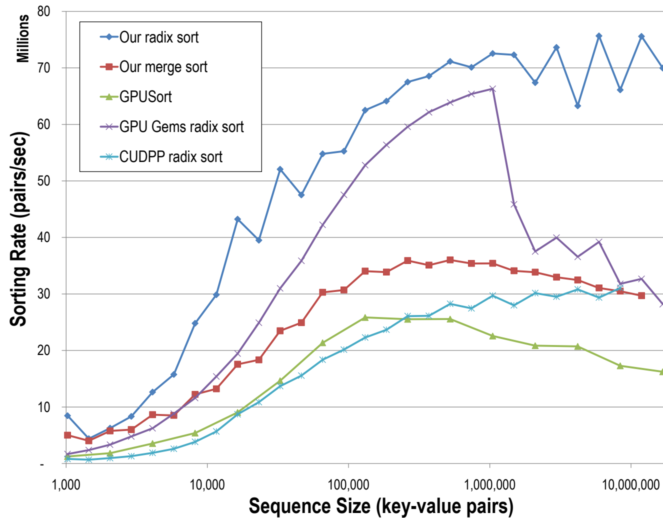


Fig. 7. Sorting rates for several GPU-based methods on an 8800 Ultra.

CUDA. Directly programming the GPU via CUDA imposes less overhead than the graphics API and exposes architectural features such as load/store to memory and the on-chip shared memory which are not available to graphics programs like GPUSort. Furthermore, the bitonic sort used by GPUSort does $O(n \log^2 n)$ work, as opposed to the more work-efficient $O(n)$ radix sort algorithms and our $O(n \log n)$ merge sort algorithm.

Our results clearly show that our radix sort code delivers substantially higher performance than all the other sorting algorithms we tested. It is faster across all input sizes and the relative performance gap increases for larger inputs. At the largest input sizes, it can sort at greater than 2 times the rate of all other algorithms and at greater than 4 times the GPUSort rate. The algorithm suggested by Le Grand [26] is competitive at array sizes up to 1M elements, at which point its sorting rate degrades substantially. This shows the importance of the block-level sorting we perform to improve scatter coherence. Our radix sort is roughly twice as fast as the CUDPP radix sort [22].

The results also show that our merge sort is roughly twice as fast as GPUSort, which is the only other comparison-based sort shown. At all but the largest input sizes, it is also faster than the CUDPP radix sort, and is competitive with Le Grand's radix sort for large input sizes. Only our radix sort routine consistently out-performs the merge sort by a substantial margin.

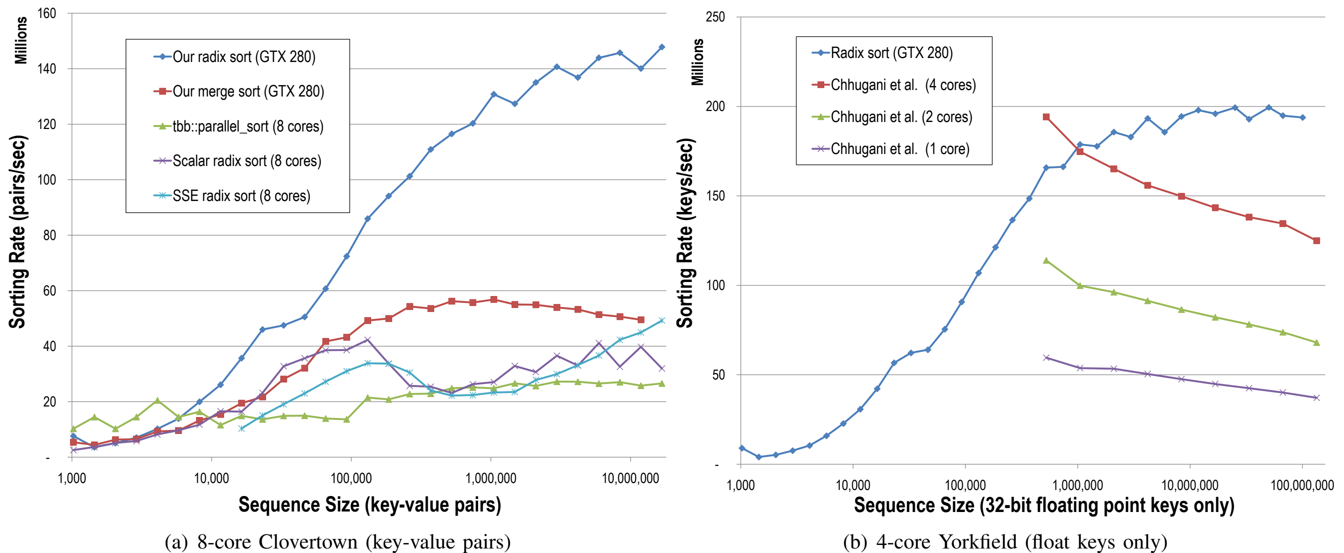
In addition to the sorts shown in Figure 7, we can also draw comparisons with two other CUDA-based partitioning sorts. The numbers reported by He *et al.* [27] for their radix sort show their sorting performance to be roughly on par with the CUDPP radix sort. Therefore, their method should perform at roughly half the speed of our radix sort and slightly slower than our merge sort. The quicksort method of Cederman and Tsigas [29] sorts key-only sequences, as opposed to the key-value sorts profiled above. We have not implemented a key-only variant of our merge sort and so cannot reliably compare its performance. However, we can compare their reported performance with our key-only radix sort results shown in Figure 8b. For uniform random sequences of 1M–16M elements, their sorting rate on GTX 280 is between 51 and 66 million keys/sec. Our key-only radix sorting rate for these sequence sizes is on average 3 times higher.

B. Comparison with CPU-based Methods

We also compare the performance of our sorting routines with high-speed multicore CPU routines. In practice, applications running on a system containing both GPU and CPU are faced with the choice of sorting data on either processor. Our experiments demonstrate that our sorting routines are highly competitive with comparable CPU routines. All other factors being equal, we conclude that applications faced with the choice of processor to use in sorting should choose the processor in whose memory the data is stored.

We benchmarked the performance of the quicksort routine (`tbb::parallel_sort`) provided by Intel's Threading Building Blocks (TBB), our own efficient TBB-based radix sort implementation, and a carefully hand-tuned radix sort implementation that uses SSE2 vector instructions and Pthreads. We ran each using 8 threads on an 8-core 2.33 GHz Intel Core2 Xeon E5345 ("Clovertown") system. Here the CPU cores are distributed between two physical sockets, each containing a multichip module with twin Core2 chips and 4MB L2 cache per chip, for a total of 8-cores and 16MB of L2 cache between all cores. Figure 8a shows the results of these tests. As we can see, the algorithms developed in this paper are highly competitive. Our radix sort produces the fastest running times for all sequences of 8K elements and larger. It is on average 4.4 times faster than `tbb::parallel_sort` for input sizes larger than 8K elements. For the same size range, it is on average 3 and 3.5 times faster than our TBB radix sort and the hand-tuned SIMD radix sort, respectively. For the two comparison-based sorts—our merge sort and `tbb::parallel_sort`—our merge sort is faster for all inputs larger than 8K elements by an average factor of 2.1.

Recently, Chhugani *et al.* [16] published results for a multicore merge sort that is highly tuned for the Intel Core2 architecture and makes extensive use of SSE2 vector instructions. This appears to be the fastest multicore sorting routine available for the Intel Core2 architecture. They report sorting times for 32-bit floating-point sequences (keys only) and benchmark performance using a 4-core Intel Q9550 ("Yorkfield") processor clocked at 3.22 GHz with 32KB L1 cache per core and a 12MB L2 cache. Figure 8b plots their results along with a modified version of our radix sort for processing 32-bit key-only floating point sequences. The combination of aggressive hand tuning and a higher-end core produces noticeably higher performance than we were able to achieve on the Clovertown system. Nevertheless, we see that our CUDA radix sort performance is on average 23% faster than the multicore CPU merge sort, sorting on average 187 million keys per second for the data shown, compared to an average of 152 million keys per second for the quad-core CPU.



(a) 8-core Clovertown (key-value pairs)

(b) 4-core Yorkfield (float keys only)

Fig. 8. Performance comparison with efficient multicore sort implementations.

VI. Conclusion

We have presented efficient algorithms for both radix sort and merge sort on manycore GPUs. Our experimental results demonstrate that our radix sort technique is the fastest published sorting algorithm for modern GPU processors and is up to 4 times more efficient than techniques that map sorting onto the graphics API. In addition to being the fastest GPU sorting technique, it is also faster than comparable sorting routines on multicore CPUs. Our merge sort provides the additional flexibility of comparison-based sorting while remaining one of the fastest sorting methods in our performance tests.

We achieve this algorithmic efficiency by concentrating as much work as possible in the fast on-chip memory provided by the NVIDIA GPU architecture and by exposing enough fine-grained parallelism to take advantage of the thousands of parallel threads supported by this architecture. We believe that these key design principles also point the way towards efficient design for manycore processors in general. When making the transition from the coarse-grained parallelism of multicore chips to the fine-grained parallelism of manycore chips, the structure of efficient algorithms changes from a largely task-parallel structure to a more data-parallel structure. This is reflected in our use of data-parallel primitives in radix sort and fine-grained merging in merge sort. The exploitation of fast memory spaces—whether implicitly cached or explicitly managed—is also a central theme for efficiency on modern processors. Consequently, we believe that the design techniques that we have explored in the context of GPUs will prove applicable to other manycore processors as well.

Starting from the algorithms that we have described, there are obviously a number of possible directions for future work. We have focused on one particular sorting problem, namely sorting sequences of word-length key-value pairs. Other important variants include sequences with long keys and/or variable length keys. In such situations, an efficient sorting routine might make somewhat different efficiency trade-offs than ours. It would also be interesting to explore out-of-core variants of our algorithms which could support sequences larger than available RAM; a natural generalization since our algorithms are already inherently designed to work on small subsequences at a time in the GPU's on-chip memory. Finally, there are other sorting algorithms whose efficient parallelization on manycore GPUs we believe should be explored, foremost among them being sample sort.

The CUDA source code for our implementations of the radix sort and merge sort algorithms described in this paper will be publicly available in the NVIDIA CUDA SDK [36] (version 2.2) as well as in the CUDA Data-Parallel Primitives Library [25].

Acknowledgement

We would like to thank Shubhabrata Sengupta for help in implementing the scan primitives used by our radix sort routines and Brian Budge for providing his SSE-based implementation of radix sort. We also thank Naga Govindaraju and Scott Le Grand for providing copies of their sort implementations.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Second ed. Boston, MA: Addison-Wesley, 1998.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second ed. MIT Press, Sep. 2001.

[3] S. G. Akl, *Parallel Sorting Algorithms*. Orlando: Academic Press, Inc., 1990.

[4] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, pp. 1–37, 2006.

[5] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," in *Proc. Eurographics '09*, Mar. 2009.

[6] V. Havran, "Heuristic ray shooting algorithms," Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov. 2000.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] B. He, W. Fang, N. K. Govindaraju, Q. Luo, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proc. 17th Int'l Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 260–269.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar/Apr 2008.

[10] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar/Apr 2008.

[11] *NVIDIA CUDA Programming Guide*, NVIDIA Corporation, Jun. 2008, version 2.0.

[12] G. E. Blelloch, *Vector models for data-parallel computing*. Cambridge, MA, USA: MIT Press, 1990.

[13] F. Gavril, "Merging with parallel processors," *Commun. ACM*, vol. 18, no. 10, pp. 588–591, 1975.

[14] T. Hagerup and C. Rub, "Optimal merging and sorting on the EREW PRAM," *Information Processing Letters*, vol. 33, pp. 181–185, 1989.

[15] D. Z. Chen, "Efficient parallel binary search on sorted arrays, with applications," *IEEE Trans. Parallel and Dist. Systems*, vol. 6, no. 4, pp. 440–445, 1995.

[16] J. Chhugani, W. Macy, A. Baransi, A. D. Nguyen, M. Hagog, S. Kumar, V. W. Lee, Y.-K. Chen, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," in *Proc. 34th Int'l Conference on Very Large Data Bases*, Aug. 2008, pp. 1313–1324.

[17] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*, Jul. 2008, pp. 16–30.

[18] P. B. Gibbons, "A more practical PRAM model," in *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 158–168.

[19] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *Proc. ACM/IEEE Conference on Supercomputing*, 1991, pp. 712–721.

[20] K. Thearling and S. Smith, "An improved supercomputer sorting benchmark," in *Proc. ACM/IEEE Conference on Supercomputing*, 1992, pp. 14–19.

[21] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin, "Fast parallel sorting under LogP: Experience with the CM-5," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 8, pp. 791–805, Aug. 1996.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*, Aug. 2007, pp. 97–106.

[23] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the Connection Machine CM-2," in *Proc. Third ACM Symposium on Parallel Algorithms and Architectures*, 1991, pp. 3–16.

[24] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007, pp. 851–876.

[25] "CUDPP: CUDA Data-Parallel Primitives Library," http://www.gpgpu.org/developer/cudpp/, 2009.

[26] S. Le Grand, "Broad-phase collision detection with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, Jul. 2007, ch. 32, pp. 697–721.

[27] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proc. ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.

[28] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conference*, vol. 32, 1968, pp. 307–314.

[29] D. Cederman and P. Tsigas, "A practical quicksort algorithm for graphics processors," in *Proc. 16th Annual European Symposium on Algorithms (ESA 2008)*, Sep. 2008, pp. 246–258.

[30] W. D. Frazer and A. C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *J. ACM*, vol. 17, no. 3, pp. 496–507, 1970.

[31] J. H. Reif and L. G. Valiant, "A logarithmic time sort for linear size networks," *J. ACM*, vol. 34, no. 1, pp. 60–76, 1987.

[32] J. S. Huang and Y. C. Chow, "Parallel sorting and data partitioning by sampling," in *Proc. Seventh IEEE Int'l Computer Software and Applications Conference*, Nov. 1983, pp. 627–631.

[33] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770–785, 1988.

[34] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics coprocessor sorting for large database management," in *Proc. 2006 ACM SIGMOD Int'l Conference on Management of Data*, 2006, pp. 325–336.

[35] A. Greß and G. Zachmann, "GPU-ABiSort: Optimal parallel sorting on stream architectures," in *Proc. 20th International Parallel and Distributed Processing Symposium*, Apr. 2006, pp. 45–54.

[36] "NVIDIA CUDA SDK," http://www.nvidia.com/cuda, 2009.