Designing Efficient Sorting Algorithms for Manycore GPUs

Nadathur Satish
University of California, Berkeley

Mark Harris Michael Garland
NVIDIA Corporation

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 reported in the literature, and is up to 4 times faster than the graphics-based GPUSort. It is also highly competitive with CPU implementations, being up to 3.5 times faster than comparable routines on an 8-core 2.33 GHz Intel Core2 Xeon system. Our merge sort is the fastest published comparison-based GPU sort and is also competitive with multi-core routines.

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 Tesla architecture and efficient data-parallel primitives, particularly parallel scan. While targeted at GPUs, these algorithms should also be well-suited for other manycore processors.

1. Introduction

Sorting is a computational building block of fundamental importance and is one of the most widely studied algorithmic problems. Many algorithms rely on the availability of efficient sorting routines as a basis for their own efficiency. Sorting itself is of central importance in applications ranging from database systems to computer graphics, and many other algorithms can be conveniently phrased in terms of sorting. 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 128 scalar processing elements per chip [19], and in contrast to earlier generations of GPUs, they can be programmed directly in C using CUDA [20, 21].

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 multi-threaded 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 [3]. We expose fine-grained parallelism in merge sort by developing an algorithm for pairwise parallel merging of sorted sequences, adapting schemes for parallel splitting [12] and binary search [4] previously described in the literature. 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. It is also highly competitive with multi-core CPU implementations, being on average 2–2.5 times

NVIDIA Technical Report NVR-2008-001, Sep. 2008.
© 2008 NVIDIA Corporation. All rights reserved.

and up to 3.5 times faster than comparable routines on an 8-core 2.33 GHz Intel Core2 Xeon system. 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. And like our radix sort, its performance compares quite favorably with a reference CPU implementation running on an 8-core system.

2. Related Work

The study of sorting techniques has a long history and countless algorithmic variants have been developed [18, 5]. Many important classes of algorithms rely on sort or sort-like primitives. Database systems make extensive use of sorting operations [9]. The construction of spatial data structures that are essential in computer graphics and geographic information systems is fundamentally a sorting process. Efficient sort routines are also a useful building block in implementing algorithms like sparse matrix multiplication and parallel programming patterns like MapReduce [7, 14].

Parallel Sorting. The importance of sorting has lead to the design of efficient sorting algorithms for a variety of parallel architectures [1]. One notable vein of research in this area has been on parallel sorting networks, of which the most frequently used is Batcher's bitonic sorting network [2]. Sorting networks are inherently parallel as they are formalized in terms of physically parallel comparator devices. Algorithms based on sorting networks are particularly attractive on platforms where data-dependent branching is expensive or impossible, since the "interconnections" between comparisons are fixed regardless of the input.

Another common approach to parallel sorting is to partition the input sequence into pieces that can be sorted independently by the available processors. The sorted subsequences must then be merged to produce the final result. A handful of efficient parallel merge techniques have been developed, including those which use elements of one sequence to break up both into contiguous runs of the output [12] and those which use parallel binary search [4].

Data-parallel techniques developed for the PRAM model and its variants are also of particular relevance to us. Blelloch [3] describes how radix sort and quicksort can be implemented using parallel scan and segmented scan primitives, respectively. Such techniques often translate well to vector machines, as demonstrated by Zagha and Blelloch [24], and are a good fit for the GPU as well.

Sorting on GPUs. Most previous attempts at designing sorting algorithms for the GPU have been made using the graphics API and the pixel shader programs that it provides. This is a highly constrained execution environment where, among other restrictions, scatter operations are not allowed and all memory regions are either read-only or write-only. Because of these restrictions, the most successful GPU sorting routines implemented via the graphics API have been based on bitonic sort [17]. The GPUSort system developed by Govindaraju *et al.* [8] is one of the best performing graphics-based sorts, although it suffers from the $O(n \log^2 n)$ work complexity typical of bitonic methods. Greß and Zachmann [11] improve the complexity of their GPU-ABiSort system to $O(n \log n)$ by using an adaptive data structure that enables merges to be done in linear time and also demonstrate that this improves the measured performance of the system as well.

Modern GPUs, supported by the CUDA software environment, provide much greater flexibility to explore a broader range of parallel algorithms. Harris *et al.* [13] and Sengupta *et al.* [22] developed efficient implementations of scan and segmented scan data-parallel primitives, using these to implement both radix sort and quicksort. Le Grand [10] proposed a radix sort algorithm using a larger radix and per-processor histograms to improve performance. He *et al.* [15] used a similar strategy to reduce scattering in their Most Significant Digit (MSD) radix sort implementation. These radix sort algorithms have been some of the fastest GPU sorting systems.

3. 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. The specific architectural details provided apply to G8x and G9x series GPUs.

Current NVIDIA GPUs are manycore chips built around an array of parallel processors [19]. A block diagram of a representative chip is shown in Figure 1. When running graphics applications, these processors execute the shader programs that have become the main building blocks for most rendering algorithms. With NVIDIA's CUDA software environment, developers may also execute programs on these parallel processors directly.

In the CUDA programming model [20, 21], an application is organized into a sequential *host* program and one or more parallel *kernels* that can be executed by the host program 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 multi-core CPUs [23].

A kernel executes 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

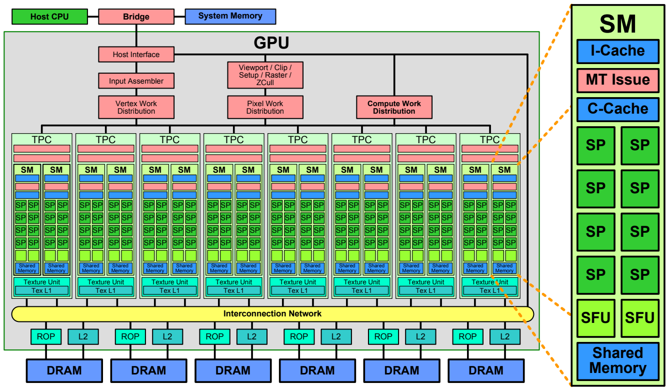


Figure 1. Block diagram of a G80 GPU with 128 scalar SP proceesors, organized in 16 SM multiprocessors, interconnected with 6 DRAM memory partitions.

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 launching a kernel, the programmer specifies both the number of blocks and the number of threads per block to be created when lauching the kernel.

Thread blocks are executed on an array of SM multithreaded multiprocessors, each of which consists of 8 SP scalar processors and is capable of executing up to 768 simultaneous concurrent threads. NVIDIA's current products range from 1 SM at the low end to 16 SMs at the high end. All thread management, including creation and scheduling, is performed entirely in hardware. The SM schedules threads to run from its pool of active blocks with zero overhead and provides for lightweight barrier synchronization costing only 1 instruction.

When designing algorithms in CUDA, it is fairly natural to think of a thread block as the rough equivalent of a CRCW PRAM of bounded size and requiring explicit barrier synchronization in place of synchronous execution. 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.

To efficiently manage its large thread population, the Tesla SM employs a SIMT (single instruction, multiple thread) architecture [19, 20]. Threads are executed in groups of 32 called *warps*. The threads of a warp are executed on separate scalar (SP) 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.

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.

Each SM is equipped with a 16KB on-chip scratchpad memory that provides a private per-block shared memory space to CUDA kernels. This shared memory has very low access latency and high bandwidth, similar to an L1 cache. Along with the SM's lightweight barriers, this memory 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. On current architectures, a warp of 32 threads performing a gather 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 threads must be live simultaneously to efficiently utilize the entire chip.

4. Radix Sort

Radix sort is one of the oldest and best-known sorting algorithms and is often amongst the most efficient for sorting small keys. Radix sort algorithms assume that the keys are represented as d-digit numbers in a radix-r notation. On binary computers, it is natural to assume that the radix $r = 2^b$ and that the keys are an integral multiple of b bits in length. The sorting algorithm itself consists of d passes which consider the i-th digits of the keys in order from least to most significant digit. In each pass, the input sequence is sorted with respect to digit i of the keys, with the requirement that this sort be stable (i.e., it preserves the relative ordering of keys with equal digits).

The sorting algorithm used within each pass of radix sort is typically a counting sort or bucket sort [5]. In a single pass, each key can be placed into one of r buckets. To compute the output index at which the element should be written—which we will refer to as the *rank* of the element—we must simply count the number of elements in lower numbered buckets plus the number of elements already in the current bucket. Having computed the rank of each element, we complete the sorting step by scattering the elements into the output array in the location determined by their ranks.

4.1. Parallelizing Radix Sort

There is substantial potential parallelism in the counting sort used for each pass of radix sort. In the simplest case, we examine 1 bit of the keys in each pass. The computation of the rank of each element can then be performed with a single parallel prefix sum, or *scan*, operation. Scans are a fundamental data-parallel primitive with many uses [3] and which can be implemented efficiently on manycore processors like the GPU [22]. This approach to radix sort was described by Blelloch [3]—who described the counting sort based on 1-bit keys as a "split" operation—and has been implemented in CUDA by Harris *et al.* [13]. An implementation is publicly available as part of the CUDA Data-Parallel Primitives (CUDPP) library [6].

This approach to implementing radix sort is conceptually quite simple. Given scan and permute primitives, it is straightforward to implement. However, it 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 being sorted. 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 scatter operations is to increase the radix r. Instead of considering 1 bit of the keys at a time, we can consider b bits at a time. This requires a bucket sort with 2^b buckets in each phase of the radix sort. To perform bucket sort in parallel, Zagha and Blelloch [24] divide the input sequence into blocks and use separate bucket histograms for each block. Separate blocks are processed by different processors, and the per-processor histograms are stored in such a way that a single scan operation gives the offsets of each bucket in each block. This enables each block to read off its set of offsets for each bucket in the third step and compute global offsets in parallel. Le Grand [10] and He *et al.* [15] 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.

4.2. Our Radix Sort Algorithm

Our approach to parallel radix sort in CUDA is also based on dividing the sequence into blocks that can be processed by independent processors. 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 radix $r > 2$ accomplishes the first goal. We accomplish the second goal by using on-chip shared memory to locally sort data blocks by the current radix-r digit. This converts scattered writes to external memory into scattered writes to on-chip memory, which is roughly 2 orders of magnitude faster.

By sorting the elements locally before computing offsets and performing the scatter, we ensure that elements with the same key within a block are present in consecutive on-chip memory locations. Because the sort is stable, elements with the same key that are consecutive within a block will go to consecutive global memory locations as well. This means that in the final scatter operation, we expect to find significant coalescing, as illustrated in Figure 4. In fact, we only expect one discontinuity in final global memory locations per bucket. By a judicious choice of $r = 2^b$, we can ensure that almost all writes are coalesced.

We implement each pass of the radix sort for the i-th least significant digit using four separate CUDA kernels.

1. Sort each block in on-chip memory according to the i-th digit using the `split` primitive (see Figure 2).

2. Compute offsets for each of the r buckets, storing them to global memory in column-major order (see Figure 3).

3. Perform a prefix sum over this table.

4. Compute the output location for each element using the prefix sum results and scatter the elements to their computed locations (Figure 4).

When sorting each block locally, we assume the current radix-r digit is a b-bit integer. We can therefore sort using b passes of the `split` operator shown in Figure 2. Recall that in CUDA, we specify the action of a single scalar thread of the thread block. Here, each thread processes one element of the sequence, receiving the current bit (true or false) of its element's key as argument `pred` to `split`. From the call to `scan`, each thread obtains the number of threads with lower IDs that have a true predicate. From this

```
__device__ unsigned int split(bool pred, unsigned int blocksize)
{
    // (1) Count 'True' predicates held by lower-numbered threads
    unsigned int true_before = scan(pred);

    // (2) Last thread calculates total number of 'False' predicates
    __shared__ unsigned int false_total;
    if(threadIdx.x == blocksize - 1)
        false_total = blocksize - (true_before + pred)
    __syncthreads();

    // (3) Compute and return the 'rank' for this thread
    if( pred )  return true_before - 1 + false_total;
    else        return threadIdx.x - true_before;
}
```

Figure 2. CUDA code for the split primitive used in block-level radix sort.

value, and the total number of true predicates, each thread can compute the rank (i.e., output location) of its elements. After b successive split and permute passes, we will have sorted all the elements of the block by the current radix-r digit.

The choice of b is determined by two competing factors. A large b leads to too many buckets present per block, and hence too many distinct global memory offsets where elements in each block need to be scattered, decreasing the coherence of the 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 the ratio of coalesced to uncoalesced memory access times on current GPU hardware, we have found $b = 4$ provides a balance between these factors.

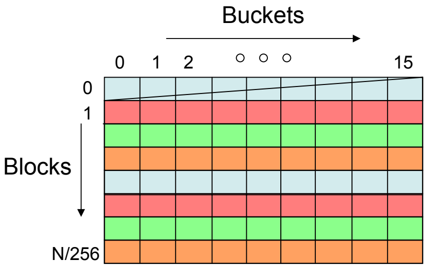


Figure 3. Per-block histogram table for computing global bucket offsets with a single scan.

The second step of our algorithm computes the number of elements in each of the 2^b buckets for each sorted block, as well as the offset of each bucket relative to the beginning of the block. The offsets are computed by finding the locations where the radix digit of an element differs from that of its neighbor. The bucket sizes are then obtained by taking the difference between adjacent bucket offsets. These are written into per-block bucket histograms as shown in Figure 3. If this table is stored in column-major order, we can obtain the global offsets for each bucket by simply performing a prefix sum operation on this table [24]. We use the efficient scan implementation provided by the CUDPP library [6].

Finally, for each element in each sorted block, we add its local offset within the block and the offset of its radix bucket in the output array to obtain the element's offset in the output array. We then write this element to the computed offset in the output array. Adjacent elements with the same radix digit in the sorted block will go to consecutive global memory locations. There are at most 2^b distinct discontinuities in these global indices per block. Therefore global memory scatters are fairly well coalesced.

Our CUDA kernels are executed by 256-thread blocks. While assigning one element per thread is a natural design decision, handling a larger number of elements per thread is actually more efficient. We process four 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.

The rules for memory coalescing on current GPU architectures are another important performance consideration. To coalesce a load or store, threads in a warp must address sequential words in memory, and the first thread's address must be aligned to 32 times the word size accessed per thread. In the scatter stage of the radix sort, it is possible for threads within a warp to write to different buckets, and thus not consecutive offsets. In fact, we expect this to be a common occurance, since we have 16 buckets and 256 threads per block, and thus an average of 16 threads per bucket. On

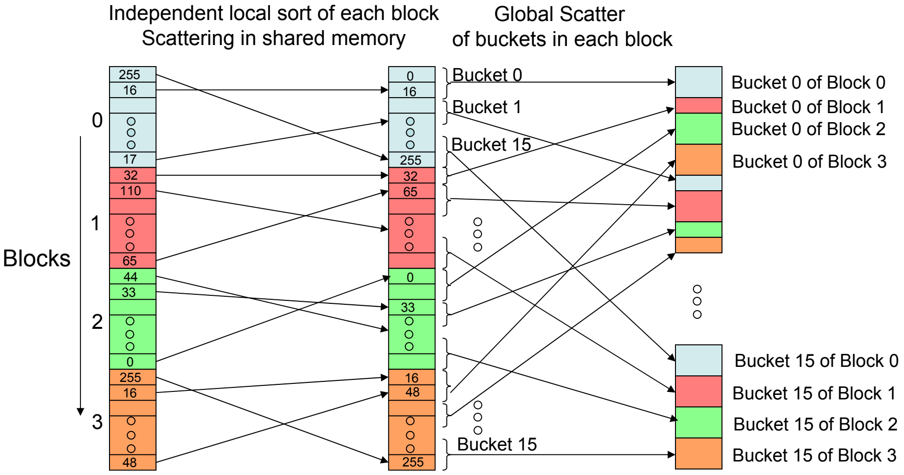


Figure 4. Scatter memory access patterns in our efficient radix sort implementation. The scattering to global memory is reduced by first doing a sort in shared memory.

the current architecture, this would prevent coalescing for almost every warp. In order to improve the performance of the code, we separate the accesses to different buckets in the same warp by iterating over the bucket boundaries in the warp. In each iteration the threads of each warp write out values belonging to only one bucket. This leads to complete coalescing of writes to global memory, but requires more iterations and more instructions than a simple uncoalesced write. This optimization could be unnecessary if future architectures provide more aggressive hardware coalescing.

5. Merge Sort

Since direct manipulation of keys as in radix sort is not always feasible, it is important to provide efficient comparison-based sorting algorithms as well. Merge sort is a divide-and-conquer algorithm that divides the input into many blocks, sorts each independently, and then merges the sorted blocks into the final sorted array. In addition to sorting as such, the merge procedure used by merge sort is often useful on its own. It can, for instance, be used for efficient online sorting, where additional elements may need to be added to an existing sorted list.

5.1. Parallelizing Merge Sort

The sorting of independent blocks of data is obviously parallel, and can be accomplished using any block-level sorting approach, including bitonic sort, radix sort [13], or quicksort [22]. Regardless of the choice of block-level sort, the main bottleneck is almost always the merging step [22]. Merging k blocks can be done in $\log k$ parallel steps using a tree of pairwise merges (see Figure 5).

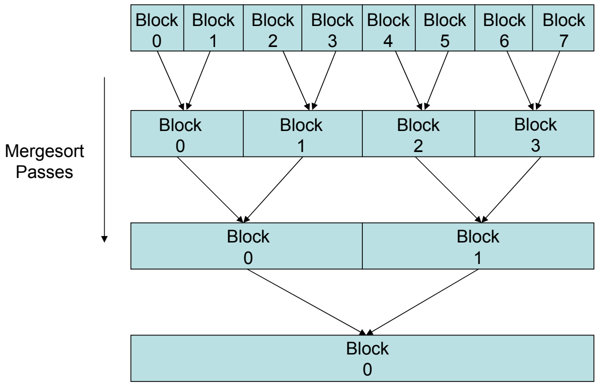


Figure 5. Tree-like pattern of merging blocks in mergesort. The number of blocks decreases and the size of each block increases as we go down the tree.

Each level of the merge tree involves the inherently parallel merge of independent pairs of blocks; 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 in designing our merge sort algorithm is therefore on exposing sufficient fine-grained parallelism within the pairwise merging of sorted blocks.

Bitonic merge [2] is one of the most commonly used parallel merging algorithms, but produces a work-inefficient $O(n \log^2 n)$ sorting algorithm. Merge sorts operating on fully sorted arrays perform the asymptotically efficient $O(n \log n)$ work, although we must still be careful that the constants involved do not make the merge slow in practice. One efficient algorithm for merging sorted blocks, proposed by Hagerup *et al.* [12], is based on partitioning the two sequences being merged using the "ranks" of a sorted array of splitting values. Here the "rank" of an array element is simply the number of values in the array less than or equal to the given element. The two sequences to be merged are partitioned using the same splitting values, so that each sub-block has a corresponding sub-block in the opposing sequence. These corresponding sub-blocks are merged independently. Hagerup *et al.* merge sub-blocks sequentially. Chen *et al.* [4] describe a parallel merging technique based on binary search. These algorithms promise to be able to efficiently merge sorted sequences in parallel without complicated data structures.

5.2. Our Merge Sort Algorithm

Our merge sort system follows the pattern outlined above. We split the input data into k blocks, sort each block in parallel, and then perform $\log k$ steps of pairwise merging (Figure 5) to produce the sorted output. We split the sequence into blocks sized to fit in the GPU's on-chip shared memory. For sorting individual blocks, we use a *bitonic sort* since it is quite efficient for small data sizes; we cannot use radix sort as our goal is to build a comparison-based sort.

As discussed above, the key to the merge sort algorithm is in developing an efficient merge algorithm. In particular, it is important to expose substantial fine-grained parallelism at all levels, rather than simply relying on the coarse-grained parallelism between merging of independent block pairs. Ideally, merging time should depend only on the total number of elements being merged and not on how these elements are divided into blocks. Such an algorithm would achieve consistent speedups at different levels of the merge tree and would be as efficient at merging the last level as the first. To expose sufficient fine-grained parallelism, we partition blocks into sub-blocks that can be merged independently (following Hagerup *et al.* [12]) and use parallel binary search to merge these sub-blocks (following Chen *et al.* [4]).

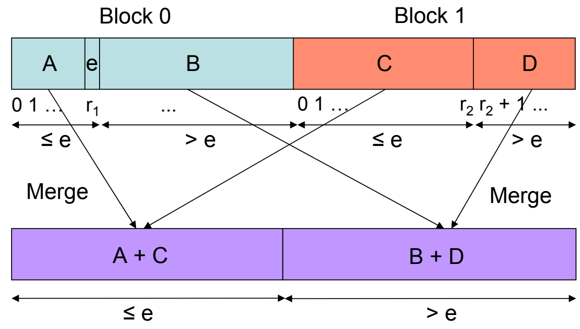


Figure 6. A two-way split of blocks based on a single element position in merge sort.

At any given level of the merge tree, we will need to merge b sorted blocks which together contain all n input elements. Since we merge pairs of blocks independently, we must simply focus on a single even/odd block pair. Consider any element e drawn from one of the blocks. If we compute the *ranks* r_1, r_2 of e in the even and odd blocks, we can split them as shown in Figure 6. We divide the even block into its first r_1 elements A and the remainder B, and divide the odd block into its first r_2 elements C and remainder D. Because we know that all elements in A, C are $\leq e$ and all elements in B, D are $> e$, we can now merge the sub-block pairs A, C and B, D independently, concatenating the results to produce the final merged block.

This only increases the available parallelism by a factor of 2. In general, we need to split each pair of blocks into multiple pieces, rather than just 2, which can be merged in parallel. We do this by sampling multiple elements with which to split the blocks. Computing the ranks of all sample elements in the even/odd pair of blocks will allow us to split the blocks into many sub-blocks: elements smaller than all the samples, those larger than just the first sample, those larger than just the first two samples, and so on. The number of sub-blocks created will be one more than the number of sample elements, and the sub-blocks may be merged independently and concatenated in a generalization of the process shown in Figure 6.

The algorithm for partitioning a block into multiple parts requires careful design to ensure that it is easy to find the ranks of each sample in both the blocks. We use every 256th element of each of the two blocks as a sample, guaranteeing that all sub-blocks have at most 256 elements. This ensures that we can merge sub-blocks in the GPU's fast on-chip shared memory, which is crucial because memory is accessed fairly randomly by the sub-block merge.

Once we select the samples, we compute their ranks in both of the even/odd blocks being merged. The rank of a sample in the block to which it belongs is trivially given by its index, leaving only its rank in the other block to be computed. To facilitate this computation, we first merge the two sequences of sorted samples. As an example, consider Figure 7. We pick elements 0 and 20 from block 0 and elements 10 and 50 from block 1 as the samples. In step 1, we merge these four samples into the sequence $\{0, 10, 20, 50\}$. In our

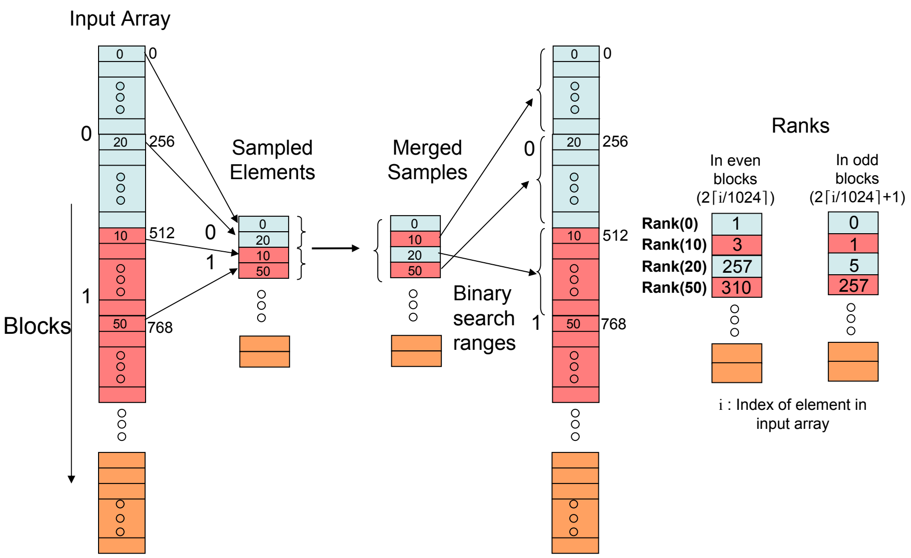


Figure 7. Splitting blocks in mergesort based on multiple input elements.

implementation, we merge samples by invoking our main merge procedure. Since the number of samples involved is small, this step takes negligible time.

Once we have the rank of each sample in the merged sample array, it is easy to compute the rank of that sample in the array of samples for the other block. It is precisely the difference between the rank of the sample in the merged sample array and its rank in its own sample array (which is already known). For example, in Figure 7, the rank of sample 20 in the samples for block 0 is 2. Since its rank in the merged sample list is 3, its rank in the samples for block 1 is $3-2=1$. This defines a narrow range of elements on which we perform a parallel binary search to compute the rank in the actual sub-block. In Figure 7, we computed the rank of sample 20 in the sample array of block 1 to be 1. This means that sample 20 is smaller than all other samples of block 1 except one (namely, element 10). We conclude that sample 20 has to lie somewhere in the first 256 elements of block 1, since if not the second sample of block 1 would also be less than 20. We can always find such a 256-element range for each sample. We can then perform a parallel binary search using one thread per sample, each thread performing a binary search over the elements of its 256-element range to find the rank. This binary search is done in the global memory of the GPU and involves irregular memory access patterns. Howver, there are just a few threads (only one per sample) and each thread only needs to do $\log_2 256 = 8$ irregular lookups for a binary search on 256 elements. Thus this step takes only a small fraction of the total execution time. Figure 7 shows the 256-element ranges for the samples and the resulting ranks.

The ranks of the samples define the sub-blocks of the odd and even blocks. Sub-blocks within a block can be naturally ordered according to the values of the elements they contain — the sub-block with the smallest elements is given the lowest number. In Figure 7, the sub-blocks are:

Sub-block #	Block 0 Elements	Block 1 Elements
1	0	-
2	1 - 2	0
3	3 - 257	1 - 4
4	257 - 310	5 - 257
...	...	...

There are two interesting facts to note about the sub-blocks we generate. The first is that we create one sub-block per every 256 elements in each block to be merged. As a consequence, every level of the merge tree will result in the same number of sub-blocks (i.e., $n/256$), leading to equal parallelism at all levels. The second interesting fact is that each sub-block of a block has at most 256 elements. This is guaranteed by the fact that no more than 256 elements in any block can lie in between two consective samples of just that block — the advantage of choosing our samples as we do. We additionally split each block on the basis of samples in the other block, but this can only decrease the size of each sub-block. This guarantees that each sub-block fits into the on-chip shared memory in the following sub-block merge step.

The final step is to merge—in parallel—corresponding sub-blocks together and concatenate the results together.

We do this using another parallel binary search, computing the rank of each element in the output merged array and then scattering values to their correct positions. The key property we exploit is that the rank of any element in the merged array is the sum of the ranks of the element in the two sub-blocks being merged. The rank of an element in its own sub-block is simply its index, since the sub-block is already sorted, and we compute its rank in the other sub-block via binary search. We use a straightforward implementation where each element is assigned its own thread, which performs a binary search over all elements of the other array to find the rank of its assigned element. In the scatter step, each thread writes out its element to the correct position. This algorithm will typically result in very irregular memory access patterns, both in the course of the parallel binary search and in the final scatter stage. As opposed to the earlier binary search when computing the ranks of each sample, this binary search performs many more irregular memory loads—there is one thread per element rather than just one per sample. This could easily be a major bottleneck. It is therefore cruicial to perform this binary search in fast on-chip shared memory. Since we have guaranteed that no sub-block can be larger than 256 elements, we are guaranteed that we can fit each sub-block into shared memory.

6. Experimental Results

We now examine the experimental performance of our sorting algorithms, focusing on comparisons with prior GPU sorting techniques and comparable multi-core CPU sorting techniques. Our performance tests are all based on sorting sequences of key-value pairs where both keys and values are 32-bit words. In brief, our experimental results demonstrate that, on current hardware, our radix sort is the fastest GPU sorting routine reported in the literature and that it is on average 2–2.5 times and up to 3.5 times faster than a comparable 8-core Intel Clovertown system. Our merge sort is the fastest published GPU sorting routine and is also competitive with the 8-core CPU routines as well.

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 PCIe 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. Even in cases where the sort itself is the entire computation, we note that the sorting rate of our radix sort is roughly 0.5 GB/s. Since PCIe bandwidth is roughly 4 GB/s and the data transfer can be overlapped with the execution of other kernels, PCIe data transfer is clearly not a bottleneck and the execution time for sort gives the clearest picture of overall sorting throughput.

6.1. Comparing GPU-based Methods.

We begin by examining the performance of several GPU sorting implementations. All GPU performance data were collected on an NVIDIA GeForce 8800 Ultra running in a PC with a 2.13 GHz Intel Core2 6400 CPU, 2GB of main memory, and using a Linux 2.6.9 kernel.

Figure 8 shows the *sorting rate* of several GPU algorithms for different input sizes. 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. The input arrays are randomly generated sequences whose lengths range from 1K elements to 16M elements, only half of which are power-of-2 sizes.

The graph shows the performance of both our sorting routines, as well as the radix sort published in *GPU Gems 3* by Le Grand [10], the radix sort algorithm implemented by Sengupta *et al.* [22] in CUDPP [6], and the bitonic sort system GPUSort of Govindaraju *et al.* [8]. 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 [11]—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 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 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 2 times the rate of all other algorithms and at nearly 4 times the GPUSort rate. The algorithm suggested by Le Grand [10] is competitive at array sizes up to 1M elements, at which point it's sort-

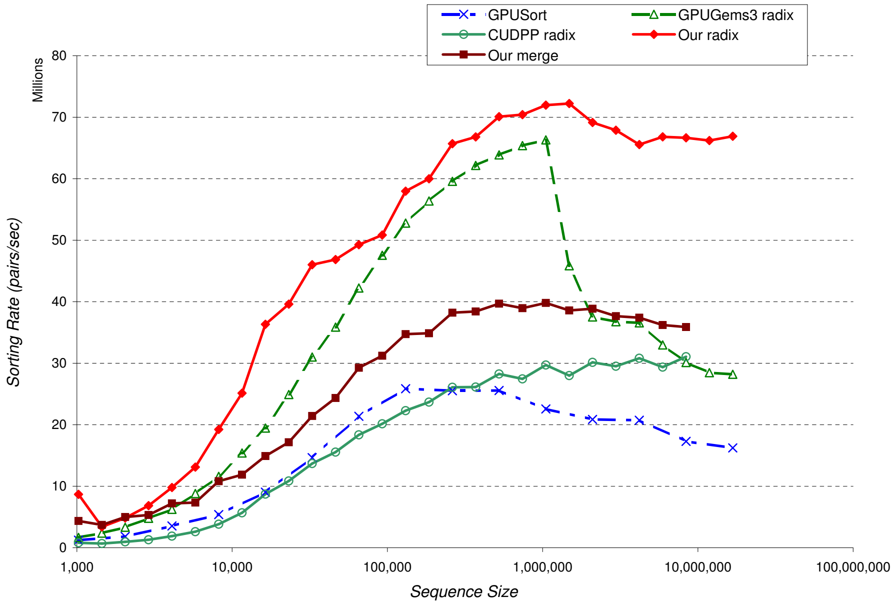
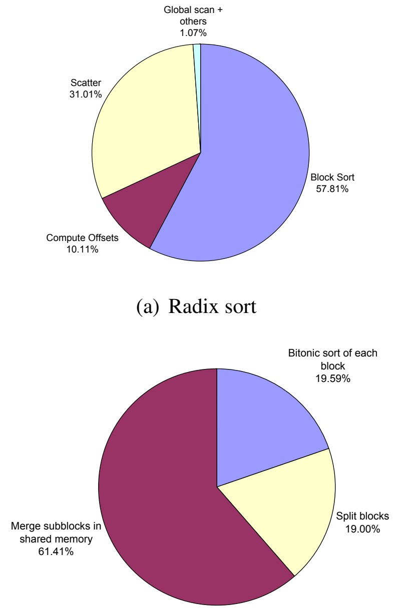


Figure 8. Sorting rates (elements sorted per second) for several GPU algorithms.

ing rate degrades substantially. This shows the importance of the block-level sorting we perform to improve scatter coherence. Based on the numbers reported by He *et al.* [15], their sorting performance is roughly on par with the CUDPP sort, making our radix sort roughly twice as fast.

The results also show that our merge sort is more efficient than all other algorithms at large input sizes, with the exception of our radix sort routine. Even at small input sizes, it is roughly twice as fast as GPUSort, which is the only other comparison-based sort, and is similarly faster than the CUDPP radix sort.

Finally, we examine the breakdown of execution times of our radix and merge sort in Figure 9. Of the three main steps in our radix sort algorithm, sorting each block in shared memory takes about 60% of the time. The scatter step, which is usually the bottleneck in radix sort algorithms, takes up only 30% of the total time. This demonstrates the efficacy of our method for improving the coherence of scattering. Block-level computation, rather than scattering to memory, is the bulk of the execution time. Since computation speed is more likely to scale with successive generations of hardware, our algorithmic approach should scale very well along with it. For merge sort, we see that blockwise merging in shared memory takes roughly 60% of the time. Again, since this is computation taking place *within* individual processors, rather than communicating with external DRAM, the performance of this algorithm should scale very well with increasingly powerful processor cores.



(a) Radix sort

(b) Merge sort

Figure 9. Breakdown of execution time for our sorting algorithms.

6.2. Comparing CPU-based Methods.

For CPU performance testing, we used an 8-core 2.33 GHz Intel Core2 Xeon E5345 system, whose archi-

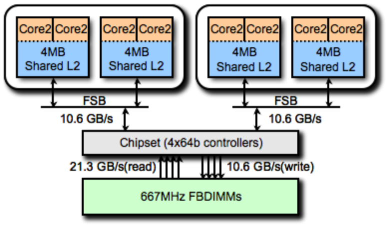


Figure 10. Diagram of the 8-core “Clovertown” Intel Xeon system used in CPU performance tests.

tecture is outlined in Figure 10. The CPU cores are distributed between two physical sockets. Each socket contains a multi-chip module with twin Core2 chips, and each chip has a 4MB L2 cache. This gives a total of 8-cores and 16MB of L2 cache between all cores. This system runs a 64-bit Linux 2.6.18 kernel.

Our baseline CPU sorting code is the `tbb::parallel_sort()` routine provided by Intel's Threading Building Blocks (TBB) [16] running across 8 threads. This is an implementation of quicksort. We also implemented our own efficient radix sort algorithm using TBB for parallelization. Finally, we tested a carefully hand-tuned radix sort implementation that uses SSE2 vector instructions and a custom Pthreads parallelization layer.

The results of our experiments are shown in Figure 11. This graph shows sorting rate performance for our two GPU sorting algorithms as well as the 3 CPU sorting implementations mentioned above. As we can see, the algorithms developed in this paper are very competitive with the CPU implementations. Our radix sort produces the fastest running times for all sequences of 8K-elements and larger. It is on average 2.6 times faster than `tbb::parallel_sort` for input sizes larger than 8K-elements. For the same size range, it is roughly 2 times faster than both our TBB radix sort and the hand-tuned SIMD radix sort. For the two comparison-based sorts—our merge sort and `tbb::parallel_sort`—our merge sort is faster for all inputs of 16K-elements and larger. Our merge sort is also competitive with the CPU radix sorts, although the relative performance factor is quite variable depending on the input size.

The results shown in Figure 11 demonstrate that the sorting rate of the quicksort-based `tbb::parallel_sort` scales relatively smoothly over input sizes, whereas the CPU radix sort performance is much more variable. We believe that this is caused by the more local nature of quicksort. After the first few partitioning steps, each core can be operating on a reasonably local set of data that will fit in its L2 cache. Radix sort, on the other hand, performs a number of global permutation steps and therefore can expect less locality. Figure 12 provides further evidence of this effect. Here we plot the relative speedup factor for 2-, 4-, 6-, and 8-thread executions of the CPU sorting algorithms over a single-threaded execution. Consistent with our expectation that quicksort should benefit more from cache locality, we see that the quicksort algorithm scales better from 1 core to 8 cores. The 8-thread quicksort execution is roughly 5.5 times faster than the single thread version for large input sizes, whereas the radix sort performance is less than 4 times faster than the single threaded performance. We do not see a similar disparity between the scaling of our radix and merge sorts in large part because the GPU relies on multithreading rather than caching to hide the latency of external DRAM. Because the GPU does not use a cache, we do not experience the same performance penalty for the scatters being performed by the radix sort.

To provide a slightly different view of performance, Figure 13 shows the parallel speed-up achieved by both CPU and GPU parallel sorts over a 1-thread execution of `tbb::parallel_sort`. We choose a single theaded version of `parallel_sort` as the base for our comparisons instead of a sequential algorithm like `std::sort` because this factors out the overhead that goes into converting a sequential to a parallel algorithm. This makes the scaling results a true measure of the parallel speedup of the algorithm. We see that all of these parallel sorts provide meaningful speed-ups. Almost all algorithms start out slower than the 8-threaded CPU `parallel_sort` algorithm at a sequence size of 1024. However, with increasing input sizes, there are more opportunities to exploit data-level parallelism. The CPU radix sort and GPU Gems 3 radix sort both seem to suffer from performance cliffs where their global scatters become too incoherent. For sequences of 1M-elements and greater, our radix sort provides by far the greatest performance boost. The GPU radix and merge sort algorithms in this paper scale, respectively, to about 14 times and 7.3 times the single-threaded `parallel_sort` performance. This is possible since modern GPUs can already execute many hundreds of parallel threads simultaneously.

7. 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 highly competitive

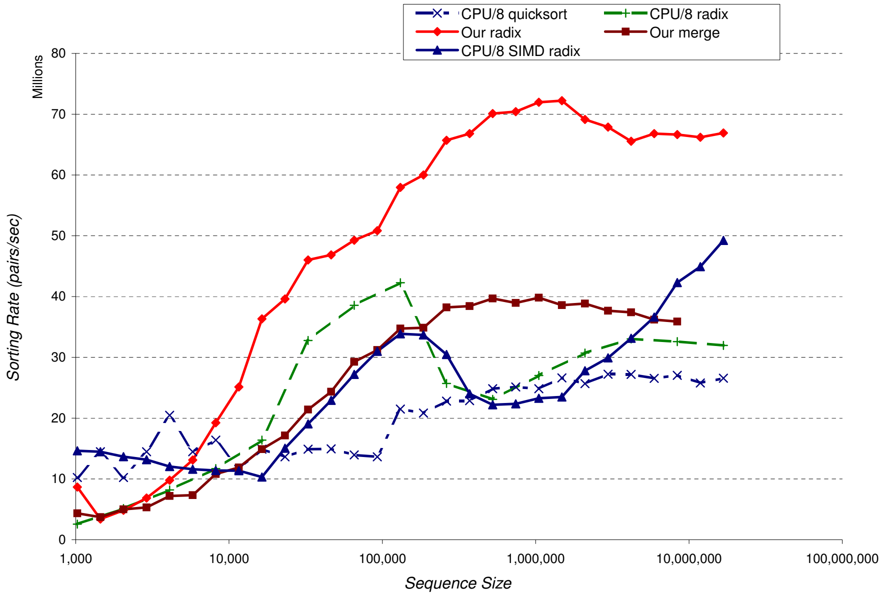
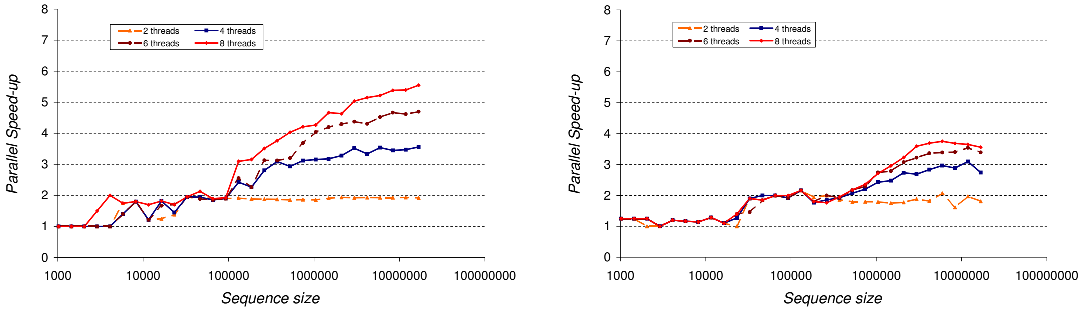


Figure 11. Sorting rates for our GPU sorts compared with an 8-core Intel Xeon system.



(a) Quicksort

(b) Radix sort

Figure 12. Parallel scaling of multi-threaded CPU sorts over single threaded execution.

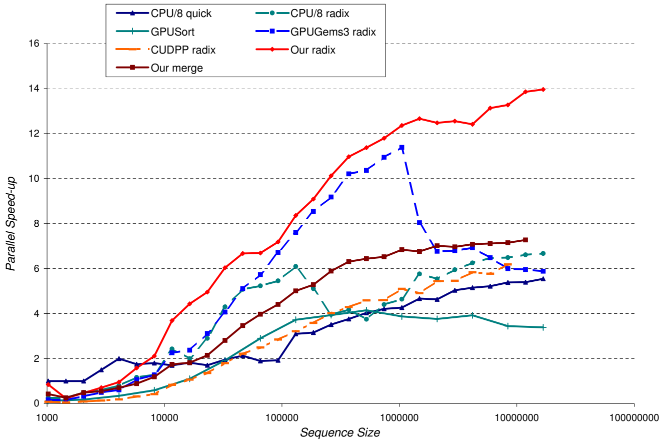


Figure 13. Scaling of sort relative to single threaded `tbb::parallel_sort` execution.

with sorting routines on multi-core CPUs, being on average 2–2.5 times and up to 3.5 times faster than comparable sorting routines on an 8-core CPU. 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 Tesla architecture and by exposing enough fine-grained parallelism to take advantage of the 1000's 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 multi-core 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 a 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 quicksort.

Acknowledgements

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.

References

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

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

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

[4] D. Z. Chen. Efficient parallel binary search on sorted arrays, with applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):440–445, 1995.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Second edition, Sept. 2001.

[6] CUDPP: CUDA data parallel primitives library. `http://www.gpgpu.org/developer/cudpp/`, Apr. 2008.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation*, Dec. 2004.

[8] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD conference*, pages 325–336, 2006.

[9] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3):10, 2006.

[10] S. L. Grand. Broad-phase collision detection with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 32. Addison-Wesley Professional, July 2007.

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

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

[13] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[14] B. He, W. Fang, N. K. Govindaraju, Q. Luo, and T. Wang. Mars: A MapReduce framework on graphics processors. Technical Report HKUST-CS07-14, Department of Computer Science and Engineering, HKUST, Nov. 2007.

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

[16] Intel threading building blocks. `http://www.threadingbuildingblocks.org`, Apr. 2008.

[17] P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 46. Addison-Wesley Professional, 2005.

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

[19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar/Apr 2008.

[21] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, Nov. 2007. Version 1.1.

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

[23] J. A. Stratton, S. S. Stone, and W. W. Hwu. MCUDA: An efficient implementation of CUDA kernels on multicores. IMPACT Technical Report IMPACT-08-01, UIUC, Feb. 2008.

[24] M. Zhaga and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the SuperComputing Conference*, pages 712–721, 1991.