Volume xx (200y), Number z, pp. 1–10

Fast 4-way parallel radix sorting on GPUs

Linh Ha, Jens Krüger, Cláudio T. Silva†

University of Utah

Abstract

Efficient sorting is a key requirement for many computer science algorithms. Acceleration of existing techniques as well as developing new sorting approaches is crucial for many realtime graphics scenarios, database systems, and numerical simulations to name just a few. It is one of the most fundamental operations to organize and filter the ever growing massive amounts of data gathered on a daily basis. While optimal sorting models for serial execution on a single processor exist, efficient parallel sorting remains a challenge. In this paper we present a hardware-optimized parallel implementation of the radix sort algorithm that results in a significant speed up over existing sorting implementations. We outperform all known GPU based sorting systems by about a factor of two and eliminate restrictions on the sorting key space. This makes our algorithm not only the fastest, but also the first general GPU sorting solution.

1. Introduction

Efficient sorting is a key requirement for many computer science algorithms, as there often exists a need to reorder input data so that it can be further explored. Today, this becomes more important than ever as the rapidly growing amount of raw, unprocessed data easily overloads the capabilities of many processing systems. In many scenarios, however, only very few input values carry the important information. Algorithms used to extract those bits and pieces frequently involve a sorting approach at their core. But not only the data preprocessing steps in the visualization pipeline require sorting algorithms also many rendering approaches and acceleration structures need some type of sorting to process the data. Many of these algorithms will not only simply run faster with improvements in the sorting subsystems but they *require* extremely fast sorting routines to allow for their use in interactive scenarios. Finally, it is worth noting that the application of ultra fast sorting is not restricted to visualization of large datasets, also many other computer graphics systems benefit tremendously from the improved performance. Only recently the area of physics based simulation in virtual environments and computer games has gained enormous attention due to its potential to dramatically increase the realism of these systems. Physics based simulation in many scenarios involves collision detection of large quantities of objects (see Figure 1), this in turn requires fast sorting of the data.

With the possibilities to increase the raw processing power of single core system seemingly coming to a halt, parallel solutions seem to be the best way for future performance improvements. Therefore, we employ the massive parallelism of current GPU architectures to sort large quantities of data at very high speed. To design an efficient sorting scheme on these systems we need to overcome two main issues:

- The first issue is how to map the efficient sequential sorting on CPUs to parallel GPU-based sorting. Although, sorting has long been studied under sequential and parallel processing models, choosing and writing an efficient sorting implementation for the GPU is still a challenge. We choose use a radix sort variant for our sorting core. Firstly, because radix sort has a linear time performance for a given range of values, which makes it superior to other approaches previously considered for GPU implementation such as merge- or bitonic-sort. Secondly, radix sort exhibits a strong dependency between successive elements as compared to previous approaches. Furthermore, our solution does not require special hardware extensions such as atomic counting as required by previous implementations [SA07] which would limit its applicability to certain

† lha@sci.utah.edu, jens@sci.utah.edu, csilva@sci.utah.edu

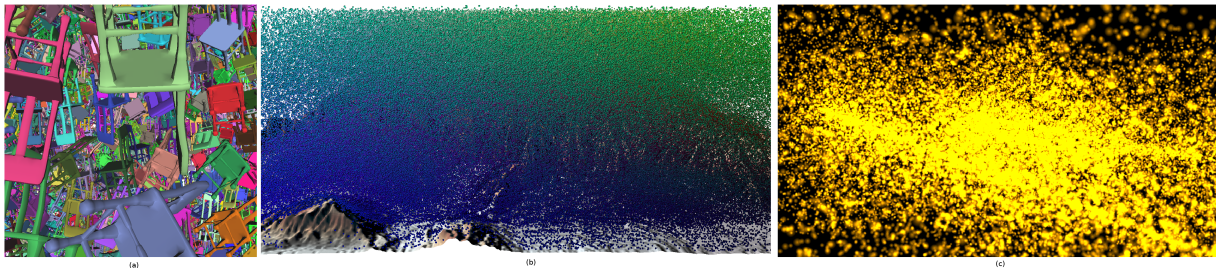

Figure 1: Various scenes with vast amounts of objects colliding with each other a) thousands of falling chairs b) millions of particles colliding with a landscape and each other and c) n-body galaxy problem. In these scenes, object sorting is a must for collision detection, requiring a large amounts of objects to be sorted at interactive frame rates.

hardware. The fast 4-way parallel radix sort we propose gives us the ability to fully exploit the parallel computational power of many GPUs. Different from previous approaches of binary-radix sort based on the CUDPP library [SHZO07], we demonstrate how to perform efficient 4-way radix (2 bits-radix). This effectively reduces the number of radix loops in half, and doubles the performance on a GPU. We also introduce a couple of improvements to minimize the overhead, and remove redundancy by a merging strategy, exploiting fast shared-memory as the computational bridge. Thereby we minimize the slow global memory access. Finally, we apply an efficient order checking strategy to be able to terminate our sorter early if we are reaching a sorted state or if the input is already sorted. This plays a particularly important role in evolving systems were the data does not change dramatically from state to state, such as the collision detection applications mentioned earlier.

- The second issue we are addressing is how to generalize the sorting strategy such that it can be applied to arbitrary types of data, for instance, negative- or floating-point-numbers, indices or pointers, and general records. This again sets us aside from previous work. Beyond that, we also present how to handle the general record sorting optimally based on index sorting, we carefully analyze the bottleneck of the problem and give a solution that can maximally exploit hardware features like instant thread switching, execution blocks, and fast random texture access of graphics memory.

In summary: To our best knowledge we present the fastest implementation of a sorter on the GPU. In addition, it is more general than other existing approaches as it eliminates restrictions on key or value length and structure. Thus, a wide variety of other graphics and non-graphics applications will benefit from this work.

The remainder of this paper is organized as follows: In Section 2 we review the previous work on GPU-based sorting. Section 3 gives an overview of our GPU sorting approach; Section 4 explains the details of the optimized CUDA realization followed by a performance evaluation in Section 5. We conclude the paper with a discussion of the results and directions of future work.

2. Related work

While CPU sorting techniques have a long history and seem to have reached their theoretical limitation as single core systems are not significantly increasing in performance, recently, parallel sorting, especially GPU based sorting, emerges as an alternative solution for this basic algorithm.

Parallel sorting networks have long been recognized as the preferred way to achieve high performance in supercomputing systems such as Cray machines. Back in 1968, Batcher [Bat68] proposed comparison-based sorting networks: the odd-even merge sort and bitonic sort. Although, having the sub-optimal complexity bound $O(n\log^2 n)$, these algorithms exploit the simplicity and symmetry to produce highly efficient parallel sorters, hence they are still widely used in parallel machines. An optimal comparison-based algorithm $O(n\log n)$ was described by Ajtai *et al.* [AKS83], however the constant hidden in the order notation is fairly large. An $O(n\log n)$ parallel sorting algorithm is also described by Leighton [Lei84] for an n-processor hypercube using random operations. Though possible, in general, a realistic $O(n\log n)$ algorithm for a parallel network is a goal not easy to achieve with comparison-based parallel sorting algorithms.

Counting based parallel sorters are an alternatives for comparison based approaches, they can lower the overall complexity to $O(n)$ or can even achieve a parallel runtime of $O(n/p)$ with p being the number of processors, however, these algorithms require inputs to be integers or values that can be mapped to integers. The most notable counting-based sorter is the parallel radix sorter. In 1991, Zagha and Blelloch implemented a parallel radix sorter on the 8-processor CRAY Y-MP vector multiprocessor machine [ZB91]. The results showed a linear scale over the number of processor and were significantly faster than a parallel quick-sort implementation on the same platform. Algorithmically, our 4-way radix is similar to this approach, however, the architecture

and programming model of CRAY machine is quite different from the one of current GPUs, making our implementation distinguishable different from their approach.

The coming of the new computing architectures such as multi-core SIMD CPUs platform or Cell processors, raised the issue of how to adapt existing parallel sorting algorithms to these new architectures to get the optimal performance out of the box. Recently, Gedik *et al.* [GBY07] implemented a bitonic sorter that was able to sort 100 million numbers in a second on the 16-SPE IBM processor machine using the Cell processor architecture. Similar performance was reported by Chhugani *et al.* [CNL*08] on the new Intel Q9550 quad-core processor. Their bitonic merge sorting implementation is highly optimized for the Intel multi-core and SSE architecture.

Our implementation targets high performance sorting on the General Processing Units (GPUs) platform. General computing on GPU (GPGPU) has achieved remarkable success, emerging as a new computing platform with high power efficiency and high scalability, being used in a variety of applications outside the scope of computer graphics and visualization, like scientific computing, geometric processing, databases, computer vision and imaging applications. For a thorough review of recent work in the field of GPGPU we refer the reader to Owens *et al.* [OLG*07, OHL*08].

Sorting on GPUs is not new and is becoming an essential component for a growing number of GPU applications. In particular for computer graphics applications, we want to keep the data inside the GPU memory and perform sorting directly on GPU rather than on CPU and transfer results back and forth between the CPU and the GPU. The early GPU based implementations by Kipfer *et al.* [KSW04] as well as their successors [GGKM06, KW05] employed the bitonic sorting algorithm. Despite the suboptimal complexity bound of $O(n\log^2 n)$ it has gained great popularity due to the fact that this sorting approach is based on a data independent sorting network making the GPU implementation easier. The complexity drawback was tackled by Gres *et al.* [GZ06] employing an adaptive bitonic sorting strategy, lowering the total complexity to the optimal bound of $O(n\log n)$ for comparison based sorters.

To further reduce the complexity, recently, counting based sorters on the GPU have gained attention. Sintorn and Assarsson [SA07] proposed a hybrid sorting algorithm based on a vectorized mergesort in combination with a bucketsort using atomic GPU operations. According to their results, the hybrid algorithm is twice as fast as the fastest previous GPU-based bitonic sort algorithm. However, their approach requires atomic functions which in turn require serial updates on the data thus wasting much of the GPU's parallel processing capabilities, consequently the performance depends heavily on the input distribution. Equally fast performance is reported by Sengupta *et al.* [SHZO07] who employed the optimized parallel prefix sum technique by Harris *et al.* [HSO07] to implement binary-radix sorting on the GPU using CUDA. Another GPU radix variation proposed by Le Grand [Gra07] also exploits the programming flexibility and controllability of the fast access GPU scratch pad memory (CUDA block shared memory) to increase the effectiveness and reduce number of radix passes over the data using a larger radix, radix-16. While giving competitive performance with uniform random inputs, the Le Grand sorting scheme seems to give the best performance with real inputs, which is illustrated in the broad phase collision framework [Gra07], and show benefits with small size of inputs. So far these three approaches are the fastest sorting schemes on the GPU outperforming all known CPU solutions.

Compared to these fastest known approaches on the GPU, our radix sorting is not only almost twice as fast but also provides a more complete sorting framework with value, index, and record sorting. Our approach can be applied not only to graphic-related problems but also to any other domain that requires fast sorting of large data.

3. Algorithm Overview

Our four-way GPU radix sorting algorithm is composed of four major subsystems:

- the order checking function
- an implicit four-way radix counting
- the prefix sum positioning
- the final mapping

We consolidate all the pseudo-code for the algorithms in this paper in the Appendix. Algorithm 1 (see Appendix) gives the implementation of a 4-way radix sorter for 32 bit keys. For the sake of simplicity we focus the discussion of the implementation on 2-bit unsigned integers while our implementation is able to handle any n-bit numerical value including floating point values. The later are first mapped to integers as proposed by Terdiman [Ter00], and then sorted in $\lceil \frac{n}{2} \rceil$ passes, each on the 2-bit radix pair, from the least significant bit to the most significant one.

At the very beginning of the sorting loop, we check the order of the current input, and immediately terminate the loop if all elements are in order. This test can be implemented very efficiently based on the optimized reduction operation as proposed by Harris [Har07]. This check is very inexpensive. It accounts only for about one-tenth of a percent of the overall sorting time, yet in many cases it greatly reduces the number of passes when the range of the input is much smaller then the full range of 32 bits, or if the array is already sorted. This is a particularly important property for many real time systems were the input array is usually close to being sorted, since it has been sorted in the last frame already.

The next step, after our algorithm made sure the array is not fully sorted already, we start the radix sorting process by computing the frequency of every element in the

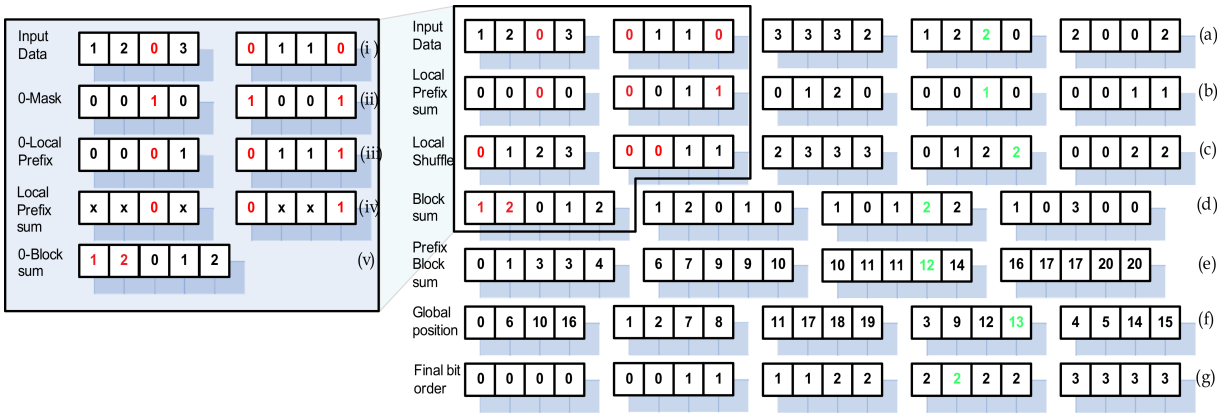

Figure 2: The basic steps of our 4-way radix sort algorithm operating on a 20 component array of 2-bit integers. The right block from top to bottom shows the steps involved in the four-way radix sorting. In the left zoom-in the computation of the local prefix sum array is shown in more detail for the digit "0".

list. To do this process in parallel, we divide the input array into blocks (see Figure 2a). The block size is determined as a multiplier of the SIMD size to exploit the full power of SIMD processing unit. Moreover, it should be large enough to hide the memory latency of global memory access and be small enough to fit onto the internal memory cache, so that all computation and memory look up can be performed with the highest possible bandwidth. However, for illustration purposes we chose the block size of four element in Figure 2.

In each of these blocks we compute the local frequency of all possible elements. As two bits are being sorted per pass we generate four counters. This counting is performed by first generating a bit mask for each of the four possible digits. To this mask we apply the shared-memory parallel optimized prefix sum computation at the same time on four counting bit combinations (see Appendix for pseudo code) and store the total count of every element in a block in the global memory (Figure 2 left zoom-in shows the computation for the "0" digit). Figure 2d depicts these counters ordered by digit. We observed that for each digit, we only concern about the local prefix result at the corresponding position of that digit in the input data, so instead of using four separated arrays to store the local prefix-sum arrays of the four digits, we use only one "local prefix sum" array with the same size as the input, and we output the corresponding local prefix of the digit, shown as red in Figure 2iii, to the local prefix sum array in Figure 2iv. This strategy greatly reduces the memory requirement of the algorithm. The local prefix sum itself, Figure 2b, indicates the order of the data in the sorted chunk of the same radix counting bit. To prevent a non-coalesced scattering effect of element wise mapping at the final state we shuffle data locally, in other words, we perform the radix sorting iteration per block. This leads to the clustering of numbers with same radix counting bit combination inside a block and enable cluster-based mapping, precondition for coalesced memory writing.

In the next step we convert the local frequency lists into global positions by computing the prefix sum over the block sum array (Figure 2e). In the final step, we use these indices to permute the values to the sorted positions. Therefore, for a digit d, in input chunk n, with a local prefix sum value m, we derive the following final position from the prefix sum block P (Figure 2e) as:

$$\text{Sorted Position} = P_d[n] + m \tag{1}$$

As an example consider the sorted position of the second "2" in the fourth chunk in Figure 2a. In this case $d = 2$, $n = 3$, $m = 1$. To compute the final position we look in the third block of the prefix bock sum (Figure 2e) at the fourth position which returns $12 = P_2[3]$ and add 1 from the corresponding local prefix sum entry (Figure 2b) to get the final index 13. This index is used to permute value from the local shuffle array (Figure 2c) into its sorted form (Figure 2g).

4. Implementation Details

While in the previous section we explained the basic algorithm of our sorter we use this section to take a closer look at the CUDA implementation of the different stages of our radix-sorting algorithm.

4.1. Blocking

The key idea of the parallel radix sort is the blocking scheme. By dividing the data into small chunks and performing many independent operations in parallel on these chunks we can efficiently use parallel architectures such as the GPU. However, the blocking should not be chosen arbitrarily. The first

thing to consider is that CUDA generally requires more than 100 threads per thread block to fully hide memory latencies. In addition, the block size should be a multiple of the *warp* size (32 on current GPUs) to guarantee coalesced reading and writing and thus achieve peak memory bandwidth over the relatively slow GPU global memory channel. Also the limitation of the shared memory (16KB on current GPUs) as well as the number of registers per block (currently 8192) set the upper limit on the number of threads inside a block. Therefore, our implementation assigns 128 threads to each block, whereas each thread handles two data inputs, hence a total of 256 elements is processed per block. Note that, none of these values are simply hard-coded into our system but based on an initialization-time query to the hardware.

4.2. Order checking

To exit the sorting loop early in case the array is sorted after processing only some of the significant bits, we employ a fast order checking algorithm in each iteration. Our implementation is based on the checking function used in the CUDPP binary radix sort library which works as follows:

- Perform parallel comparison between the current element with the next data elements, write out result to the comparison array, 0 if in correct order, 1 other wise.
- Perform parallel reduction sum on the comparison array as proposed by Harris [Har07]. If the final result is 0 then input data is in order, otherwise not in order.

While the second step is already extremely well optimized, the first step, has a couple of drawbacks in the reference implementation. Firstly, each thread has to read two consecutive elements thus every value is read twice and the GPU is unable to perform coalesced reading, which is very important for achieving peak performance. Secondly, the first step writes the result of the comparison into relatively slow global memory while in the second step that very data is read back into shared memory to perform the reduction.

By successfully addressing the above mentioned issues we were able to *triple* the performance of the order checking routine. Firstly, we read only one element per thread into shared memory thus allowing for coalesced reading and cutting the read operations in half. Secondly, we perform a partly reduce-add, the parallel reduction sum operating on block-based data, in shared memory and write only the single block sum into global memory for the second step. Algorithm 3 in the Appendix shows the pseudo code of parallel test function.

While this parallel order checking approach is very fast it still is an atomic operation that always executes completely before returning a decision. In contrast to this, a serial order checking approach would stop once the first out of order element is found. To approximate this behavior, we first perform the test on the first n' elements of the entire array of n elements. If this quick test fails we perform the next sorting step and a complete test is avoided. If we can not find any out of order elements in the quick test we perform the check on the remaining $1+n-n'$ including the last element of the n' set. This approach introduces no measurable overhead in the worst case scenario where the subset test does not detect out of order elements, but it significantly improves the runtime otherwise as our experiments have shown.

4.3. 4-way prefix sum

Algorithm 2 in the appendix shows the pseudo code of 4-way prefix sum implementation. Our implementation extends the reference CUDPP implementation. Figure 2 left illustrates the algorithm output with radix combination value the 0. We concurrently generate the binary mask for each radix combination (ii), compute the local prefix sum of the mask (iii), merge the prefix sum's results for all combinations (iv) and send the total number of each radix combination in each block to the output block sum array (v).

4.4. Global counting

The 4-way prefix sum algorithm outputs the local counting result of each block that shows the sorted order of elements inside the block. To compute the global order, we have to know the position of the first similar element of that block, that is the total number of similar element of previous blocks. Once again, this information can be computed by a prefix sum over the block sum of each individual radix combination, but in this case, we do not need four-way prefix sum, since it yields the same results with one-way prefix sum when we layout the block sum output of 4-bit combination sequentially. For optimal performance we use the fully optimized version ofthe prefix sum provided in the latest CUDPP release [HOS*07].

4.5. Positioning and final mapping

As mentioned earlier, the global position is computed by adding the local block position to the prefix block sum of each bit combination. The resulting fetch returns the new position of the data in the sorted array, a simple mapping function will then permute the data. Similar to the full order checking operation, we only need the position to perform the mapping function, so we can combine these two steps into one, and compute the global position value in shared memory only without explicitly moving it into global memory, this strategy saves us one read/write operation into the global position array which is costly. Overall, this improvement alone results in a 20% speedup.

4.6. Local shuffling and coalesced mapping

The major weakness of the traditional radix sorting is the final shuffle step. Since radix sorting shuffles data globally in a random-like pattern, the shuffling is cache-unfriendly.

This explains why radix sorting may be slower than other cache-friendly sorts like quick-sort, with small input size, despite its lower complexity bound $O(n)$ vs $O(n \log n)$ of quick-sort. Though there is no cache strategy implemented on the GPU's global memory, random-like scattering patterns lead to non-coalesced writing, reducing the efficiency of GPU radix sorting significantly, and accounting up to 36% of the sorting time.

To address this issue, we propose in-block shuffle radix sorting. The in-block shuffle radix sorting strategy is based on the observation that the Least Significant Byte (LSD) radix sorting is stable, it preserves the order of numbers with the same radix combination bits in a block, that means consecutive numbers of the same radix sorting bits in a block will move together to successive positions in the output. Thus, if we perform local radix shuffling inside a block, it will produce clusters of the same radix combination bits in sorted order, the number of a cluster in a block is at most the number of the combination. The numbers in the same cluster will move together to the new location in the global shuffle step, this is likely to produce the needed condition for coalesced writing.

The in-block shuffle radix sorting procedure is similar to an implicit four-way radix counting with an additional step: in-block shuffle, in which we reorder the number in the shared memory before we output the result into global memory. Since in-block shuffle was performed in the shared memory, it is fast, efficient, and no longer constrained by the coalesced writing condition. Moreover, since the mapping position of a number in a cluster can be defined implicitly based on the mapping position of the first number in the cluster, we do not need to store all the mapping positions but only the first one in a cluster. This consequently, reduces the memory footprint needed by traditional radix sorting approach.

To fully utilize the available bandwidth, we modified the non-coalesced element-wise mapping in the shuffle steps to coalesced cluster-wise mapping. To accomplish this we have to satisfy the coalesced writing condition that the i^{th} threads will output the data to the i^{th} position in 128-byte aligned region. In this case, since we know the mapping position, we can determine the corresponding thread. We exploit shared memory to achieve both coalesced reading and writing. The mapping involves three steps, as shown in Algorithm 4: (1) read data in block to shared memory, (2) compute the position of first elements in a cluster and corresponding thread, (3) each thread outputs corresponding data to global memory output. We perform the last two steps in parallel with all threads, once for each radix combination.

Overall the modified version with in-block shuffle radix counting and coalesced cluster-wise mapping process gives a 50% improvement in the performance on graphics cards that support CUDA 1.0 and 1.1, e.g. G80 and Geforce Quadro FX generation. And even on the newer cards, e.g. GTX 280, it still gives a 10% improvement.

4.7. Index and Record Sorting

The sorting value alone is hardly of any use for real applications. What we need is the sorted index, or data pointer to determine the related information. The main difference between value and index sorting is that index sorting requires an additional shuffle step of the indices. As a result, index sorting requires higher memory bandwidth, and exacerbates the effect of cache-misses.

Normally the index is not included in the original data, so for each input value we have to generate a companion index indicating the position of the value in the original input array. In its simplest form, the index array is a sequence of number from 0 to $n-1$. Index sorting only requires operations on the key value and its accompanied index, other related information can be traced back from the original array using the sorted index.

It is not obvious from other GPU sorting methods how to extend the value sorting frameworks to such an index sorting, as often, only the timings for sorting the keys are given but not the timings for the entire sorting pipeline of rearranging the associated records. A typical strategy is to combine the sorting value and the index into the same sorting register and only consider the bits of the sorting value during the actual sort. While this idea can be integrated into very much every sorting framework without impacting the performance, there are several disadvantages to this approach:

- The number of bits available for the position being sorted is reduced, which—in the case of physical based simulation like a collision detection system—results in a significant loss in precision;
- The number of bits available for index range is also significantly reduced, limiting the number of sorting objects and thus putting the use of the GPU—which is usually most efficient on large data—in question.

With our radix sort approach, however, the key counting process is independent from the value of related indices, we simply generate the permutation array and finally shuffle the indices or data pointer with this array. Thus, with the current generation of 32 bit GPUs up to 4 billion key-index pairs or records can be sorted in theory.

While index sorting is sufficient in most applications, sometime a full record sorting is desired, especially to increase the cache coherency with frequently-accessed sorted data. In comparison to index sorting, record sorting can be implemented similarly by first generating a simple spatial index for each input value, performing index sorting on the key-index pair array, and then perform the full shuffling step for record sorting if needed. Because the sorted index array is a permutation of the index sequence, as we perform

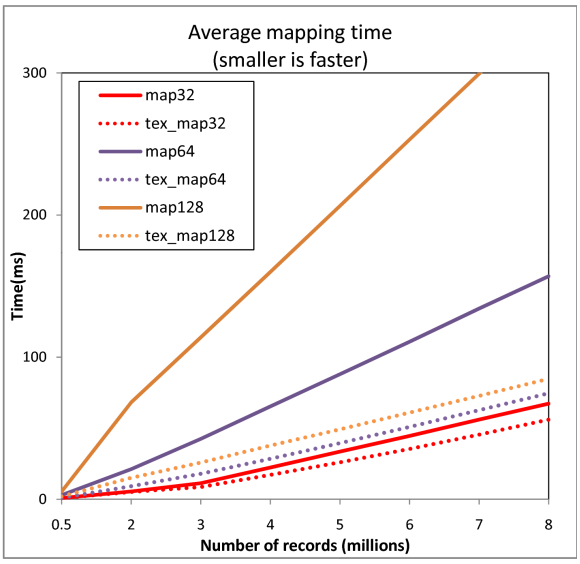

Figure 3: Record shuffling with typical size of record 32, 64 and 128 bits. We break regular array of record structure to structure of array with typical length types. The figure shows the benefit of using texture to improve shuffling rate versus normal mapping using global memory

the final shuffle step, we have to access data in arbitrary-like order from the original data array while we can output sorted data sequentially. The whole process leads to non-coalesced reading—coalesced writing pattern on the GPU memory. Consequentially, the reading time for the final shuffle step could become a major bottleneck in the performance of record sorting. Fortunately, limited bandwidth of the non-coalesced reading can be significantly improved using texture memory, with uniformly distributed inputs we observe the factor of three in the shuffling performance by using texture fetching over the typical mapping using global memory, as shown on the Figure 3.

5. Results

In the following we compare our implementation to the fastest GPU and CPU sorters in existence today: The optimized CPU RadixSort as proposed by Herf [Her01], the STL sort [Col08], the multi-threaded TBB parallel quick-sort by Intel [Int08], the GPU binary-radix sort from the the CUDA Data Parallel Primitives Library [HOS*07], the radix-16 sort by Le Grand [Gra07], and an eight-way parallel version of our radix sort. All test were performed on an Opteron AMD 275 quad dual-core 2.2Ghz, system with 6 GB of memory, and 1024K L1 cache equipped with an NVIDIA Geforce 8800 GTX.

The timings were performed with varying number of inputs ranging from 0.5M to 16M elements with uniform and Gaussian distribution. The running time measured in milliseconds is averaged over 100 runs, with the inputs re-drawn randomly for each run. The running times do not include the data transfering time between CPU and GPU because the bandwidth between CPU and GPU is low, making transferring process a performance bottleneck. Moreover, we want to exploit GPU sorting for GPU applications, in that case the data are already available in the graphic memory.

As can be seen in Figure 4, our coalesced 4-way radix sort gives the best performance, both, with uniform and Gaussian distribution inputs. It is between 1.5 and 2.2 times faster than the next best GPU sorting result: the radix-16 that also running on CUDA by Le Grand. Figure 4 also shows a speedup of 1.5 from the 4-way non-coalesced mapping radix to the final version of 4-way radix, it demonstrates the impact of coalesced access on the performance of CUDA GPU implementation. On our system, the fastest CPU sorting implementation is the optimized CPU radix sort, however, even this sorter is still 1.5-2.2 times slower than our GPU radix implementation. Expensive thread creation explains why Intel TBB sorting does not give competitive performance on our system.

In comparison to the performance of value sorting only, we experience a reduction by a factor between 1.5 and 1.8 of the typical radix index sorting approach with the same value input array, which we attribute mainly to the non-coalesced shuffling pattern. As we apply the coalesced radix index sorting scheme, as discussed on Section 4.6, the performance only suffers a negligible factor of about 1.1. Since only the shuffling process is impacted by the non-coalesced memory access pattern, the coalesced scheme results in a much more stable performance than the arbitrary non-coalesced sorting approach.

To confirm that our choice of a 4-way radix sort implementation is optimal at least on the current hardware, we implement a similar framework with 8-way radix sorting scheme. The timing result with 8-way radix value sorting is shown in Figure 4. From the timings, our 8-way radix GPU with value sorting, is about 5%-10% slower than 4-way radix. The 8-way radix index sorting approach using the scan framework is currently not feasible due to limited amount of shared memory available on the current generation of graphics cards. As this may be different on the next generation hardware, we realized our entire algorithm as a template with the parallelization factor as the template parameter. This way we can embed our sorting algorithm in an automatic tuning program, that on a new system runs a calibration procedure consisting of a few sorts to select the best approach for the actual hardware it is running on.

To test the scalability of our approach on future devices, we ran our sorting framework on the latest GPU platform an NVIDIA GTX 280. The results from the Figure 5 clearly show that our radix algorithm scales well with the latest generation of GPUs. As the memory bandwidth offered by GTX

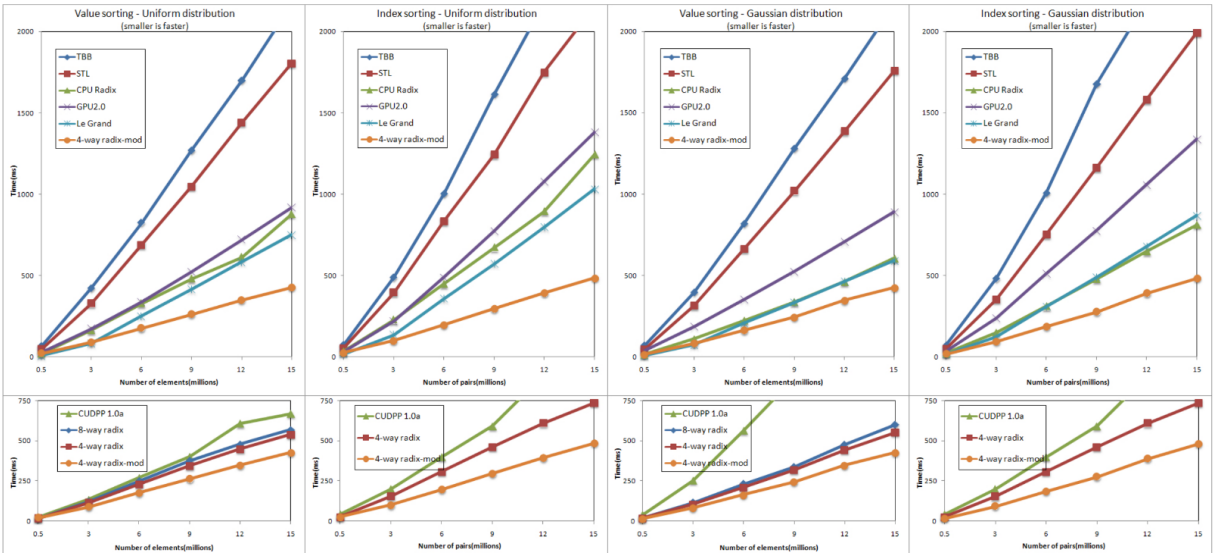

Figure 4: Average sorting runtime for varying input sizes with different sorting strategies, with uniform distribution and Gaussian distribution input, with value sorting and index sorting .

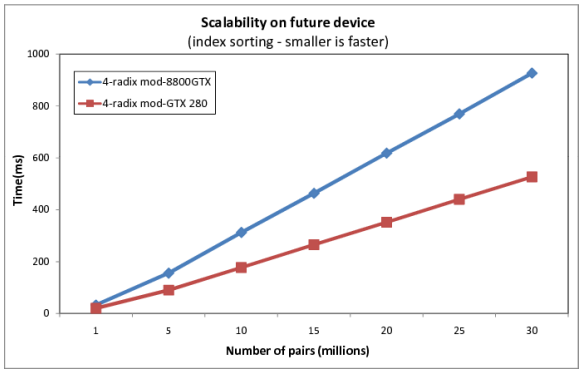

Figure 5: Performance on the newest generation of NVIDIA GPUs, the Geforce 280GTX. It can be seen clearly that as the memory bandwidth doubles so does our sorting performance

280 doubles the available bandwidth of GTX 8800 GTX, it doubles the performance of our sorting approach.

6. Applications

To illustrate the importance and benefit of GPU sorting framework on practical problem, we applied our framework to a GPU broad-phase collision system.

6.1. Broad-phase collision detection

Our collision detection system builds on the proven sweep and prune strategy, a broad phase algorithm developed by Baraff [Bar92] and then implemented by Cohen *et al.* [CLMP95]. The method is mainly used for interactive and exact collision detection in a large-scale complex environments, such as what are shown in Figure 1. The core function and also the main performance bottleneck is a sorting subsystem that first sorts the projected extents of objects' bounding boxes on a cartesian axis, then use a second pass over the sorted list to determine overlaps between these boxes.

In our implementation we first compute the axis aligned bounding boxes (AABBs) of the objects. We have chosen AABBs since they can be efficiently computed on GPU using parallel reduction max and min on the input meshes. From the AABBs a list of records is generated. Every AABB contributes two records: the max/min AABB-coordinates combined with an index to the object, and a bit indicating if this is the max or the min coordinate. This list is sorted with respect to the x-coordinate and scanned for out of order indices in parallel. Where out of order means that we scan for a min index of one bounding box not being followed by the max index of the same AABB but by an index of another AABB. From the AABBs associated with the out of order indices a new list of AABBs which overlap in x projection is generated. By using the min/max bit we make sure that every collision is only detected once. This entire process is repeated in y- and z-direction using the overlap list of the previous scan, finally only the small list of actually overlapping AABBs is returned for narrow-phase collision detection an the CPU. We accelerate this algorithm by embedding our novel optimized GPU-based radix sorting system as the core algorithm. To further improve the performance of the overall

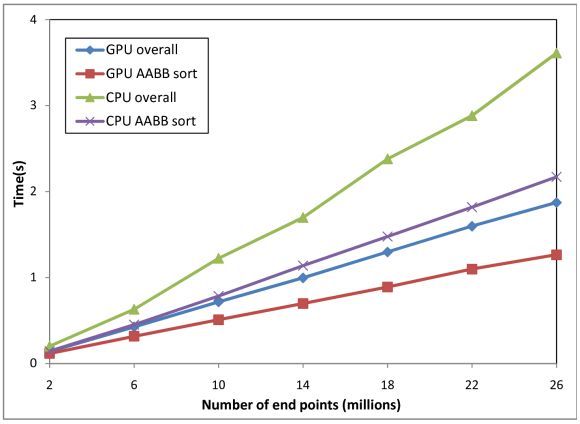

Figure 6: Timings result for the GPU based collision detection system and the embedded 4-way radix sorter, compared to an optimized CPU system also based on a radix sorter

system we also implement other functions in the framework with its optimized parallel version on GPU. The AABB sorting in the framework is a variation of index sorting scheme, in that each value is corresponding to one end value of an object's bounding box stored together with the index of the object.

In Figure 6 we present timings for our entire GPU broad-phase collision detection system on an NVIDIA Quadro FX 5600 graphics card with 1.5 GB local video memory. We compare this system to a highly optimized CPU implementation that uses an optimized radix sort implementation. The CPU approach makes use of all the cores of our quad dual-core AMD Opteron 275 processor with 6 GB memory.

As can be seen in Figure 6, sorting is the most costly operation in the algorithm as it accounts for about 70% of the overall performance. Figure 6 also gives a comparison of our approach to the highly optimized CPU sorter showing a 1.5x to 2.0x speedup depending on the data size; as usual the GPU becomes more efficient with increasing data size.

7. Discussion and Future Work

In this paper we have presented a GPU linear-time sorting framework that outperforms previously published methods. At the same time we lift the previously imposed limitations on the element count and sorting precision of a GPU based sorter. Thus, we provide a complete solution for the sorting problem, making our solution superior to CPU based approaches.

While we improved the current state of the art in sorting, there is still room for further improvement: A more efficient histogram and counting sort could make sorting more than two bits a time more efficient, and a hybrid input-adaptive framework that will select the best sorting method for each type of inputs and input size.

In the future we will also investigate novel applications for our GPU based radix sorter outside of the computer graphics area, such as data mining applications. For these applications additional out of (GPU-) core sorting approaches and GPU cluster solutions may be required to handle extremely very large data sets in reasonable time.

Acknowledgments. We would like to thank Huy T. Vo, and the anonymous reviewers for insightful discussions and constructive comments that helped us to substantially improve this paper. This research has been funded the National Science Foundation (grants CNS-0751152, CCF-0528201, OCE-0424602, CNS-0514485, IIS-0513692, CCF-0401498, OISE-0405402, CNS-0551724), the Department of Energy SciDAC (VACET and SDM centers), and IBM Faculty Awards (2005, 2006, and 2007); It also was made possible in part by the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10. L. Ha was partially supported by the Vietnam Education Foundation fellowship.

References

[AKS83] AJTAI M., KOMLÓS J., SZEMERÉDI E.: An 0(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing* (1983), pp. 1–9.

[Bar92] BARAFF D.: *Dynamic simulation of nonpenetrating rigid bodies*. PhD thesis, Cornell University, Ithaca, NY, USA, 1992.

[Bat68] BATCHER K. E.: Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32* (1968), pp. 307–314.

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M. K.: I-collide: An interactive and exact collision detection system for large-scale environments. In *In Proc. of ACM Interactive 3D Graphics Conference* (1995), pp. 189–196.

[CNL*08] CHHUGANI J., NGUYEN A. D., LEE V. W., MACY W., HAGOG M., CHEN Y.-K., BARANSI A., KUMAR S., DUBEY P.: Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow. 1*, 2 (2008), 1313–1324.

[Col08] COLLECTION G. C.: Standard template library. http://gcc.gnu.org/, 2008.

[GBY07] GEDIK B., BORDAWEKAR R. R., YU P. S.: Cellsort: high performance sorting on the cell processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 1286–1297.

[GGKM06] GOVINDARAJU N., GRAY J., KUMAR R., MANOCHA D.: GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 325–336.

[Gra07] GRAND S. L.: Broad-phase collision detection with cuda. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, Aug. 2007.

[GZ06] GRESS A., ZACHMANN G.: GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Rhodes Island, Greece, 25–29 April 2006).

[Har07] HARRIS M.: Optimizing parallel reduction in cuda. http://tinyurl.com/6dazkd/reduction.pdf, 2007.

[Her01] HERF M.: Radix tricks. http://www.stereopsis.com/radix.html, 2001.

[HOS*07] HARRIS M., OWENS J., SENGUPTA S., ZHANG Y., DAVIDSON A.: Cudpp: Cuda data parallel primitives library. http://www.gpgpu.org/developer/cudpp/, 2007.

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with cuda. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, Aug. 2007.

[Int08] INTEL: Intel thread building blocks 2.1. http://www.threadingbuildingblocks.org/, 2008.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: UberFlow: a GPU-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 115–122.

[KW05] KIPFER P., WESTERMANN R.: *Improved GPU Sorting*. Addison-Wesley, 2005, ch. 46, pp. 733–746.

[Lei84] LEIGHTON T.: Tight bounds on the complexity of parallel sorting. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1984), ACM, pp. 71–80.

[OHL*08] OWENS J. D., HOUSTON M., LUEBKE D., GREEN S., STONE J. E., PHILLIPS J. C.: GPU computing. *Proceedings of the IEEE 96*, 5 (2008), 879–899.

[OLG*07] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[SA07] SINTORN E., ASSARSSON U.: Fast parallel GPU-sorting using a hybrid algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* (2007).

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *Graphics Hardware 2007* (Aug. 2007), ACM, pp. 97–106.

[Ter00] TERDIMAN P.: Radix sort revisited. http://tinyurl.com/6l6p2k, 2000.

[ZB91] ZAGHA M., BLELLOCH G. E.: Radix sort for vector multiprocessors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1991), pp. 712–721.

Appendix A: Pseudocode

Algorithm 1 Four-way radix sorting

```
blockSize ← 256
Allocate and initialize 4 block-sum array on GPU mem
for bit = 0 to 30 do
    Perform order checking on the current stage
    if elements are in order then
        break;
    end if
    Divide input data into equal blocks
    for all blocks in parallel do
        Perform shared-memory 4-way scan on each block
        Output total number of each scan path to block sum array
    end for
    Perform scan prefix sum on the block sum array
    Compute the new position for elements of radix sort
    Map the element to the right position
    bit ← bit + 2
end for
```

Algorithm 2 4-way prefix sum with in-block shuffle

```
Allocate 4 counting arrays, cnt[4]
Read data in to shared-memory block s_data
for all threadId in thread block do
    Extract 2 bits combination from the input data[id]
    for b = 0 to 3 do
        cnt[b][threadId] ← (b == extract_bits)
    end for
    Built four way sum tree, Algorithm 5
    Clear the last element of each counting array
    Down sweep the tree, scan in place Algorithm 6
    synchthread
    Shuffle data in the shared memory
    s_data[cnt[extract_bits][threadId]] ← data
    synchthread
    Output local sorted data to global memory
end for
```

Algorithm 3 Parallel order checking

```
Allocate shared memory for each thread block
//Compute elements id
id ← threadIdx.x + threadIdx.y * blockSize
for all threads inside thread block do
    //Reading one value per thread to the shared memory
    shared[threadIdx.x] ← data[tid];
    if threadIdx.x = 0 then
        Read the next element to the last data inside block
    end if
end for
//Wait for all threads to finish reading
syncthreads()
//Perform order checking
shared[id] ← (shared[i] > shared[i + 1]);
Perform optimized reduction on shared array
Write out reduction result to global array
```

Appendix A: Pseudocode

Algorithm 4 Coalesced block mapping for the n chunk input

if $threadIdx.x = 0$ **then**
  Load block count and block prefix sum of each counting bits
end if
for all threads inside thread block **do**
  Load the block-sorted data to the block shared-memory
end for
syncthreads()
for d = 0 to 3 **do**
  Determine the first position of output block: the block prefix sum value $P_d[n]$
  Find the first aligned coalesced position on the output
  Aligned_pos = $P_d[n]$ - $P_d[n]\%warp\_size$
  for all threads inside thread block **do**
    Data access position $m \leftarrow threadId - P_d[n]\%warp\_size$
    $SortedPosition \leftarrow Aligned\_pos + threadId$
    if Sorted Position is in the output range **then**
      Map the data from shared memory position m to global memory
    end if
  end for
end for

Algorithm 5 Build 4 ways sum tree

for $d = 0$ to $\log_2 n - 1$ **do**
  for all i=0 to $(n-1)/2^{d+1}$ in parallel **do**
    #pragma unroll
    for b = 0 to 3 **do**
      $cnt[b][i + 2^{d+1} - 1] \leftarrow cnt[b][i + 2^d - 1] + cnt[b][i + 2^{d+1} - 1]$
    end for
  end for
end for

Algorithm 6 Down-Sweep 4 ways

for b = 0 to 3 **do**
  $count[b][n-1] \leftarrow 0$
end for
for $d = \log_2 n - 1$ downto 0 **do**
  for all i=0 to $(n-1)/2^{d+1}$ in parallel **do**
    #pragma unroll
    for b = 0 to 3 **do**
      $t \leftarrow count[b][i + 2^d - 1]$
      $cnt[b][i + 2^d - 1] \leftarrow cnt[b][i + 2^{d+1} - 1]$
      $cnt[b][i + 2^{d+1} - 1] \leftarrow t + cnt[b][i + 2^{d+1} - 1]$
    end for
  end for
end for