Fast Parallel GPU-Sorting Using a Hybrid Algorithm

Erik Sintorn
Department of Computer Science and Engineering
Chalmers University Of Technology
Gothenburg, Sweden
Email: erik.sintorn@chalmers.se

Ulf Assarsson
Department of Computer Science and Engineering
Chalmers University Of Technology
Gothenburg, Sweden
Email: uffe at chalmers dot se

***Abstract*— This paper presents an algorithm for fast sorting of large lists using modern GPUs. The method achieves high speed by efficiently utilizing the parallelism of the GPU throughout the whole algorithm. Initially, a parallel bucketsort splits the list into enough sublists then to be sorted in parallel using merge-sort. The parallel bucketsort, implemented in NVIDIA's CUDA, utilizes the synchronization mechanisms, such as atomic increment, that is available on modern GPUs. The mergesort requires scattered writing, which is exposed by CUDA and ATI's Data Parallel Virtual Machine[1]. For lists with more than 512k elements, the algorithm performs better than the bitonic sort algorithms, which have been considered to be the fastest for GPU sorting, and is more than twice as fast for 8M elements. It is 6-14 times faster than single CPU quicksort for 1-8M elements respectively. In addition, the new GPU-algorithm sorts on $n \log n$ time as opposed to the standard $n(\log n)^2$ for bitonic sort. Recently, it was shown how to implement GPU-based radix-sort, of complexity $n \log n$, to outperform bitonic sort. That algorithm is, however, still up to $\sim 40\%$ slower for 8M elements than the hybrid algorithm presented in this paper. GPU-sorting is memory bound and a key to the high performance is that the mergesort works on groups of four-float values to lower the number of memory fetches. Finally, we demonstrate the performance on sorting vertex distances for two large 3D-models; a key in for instance achieving correct transparency.**

I. Introduction

Sorting is a general problem in computer science. Bitonic sort has primarily been used by previous GPU sorting algorithms even though the classic complexity is of $n(\log n)^2$ [2], [3]. It is, however, possible to modify bitonic sort to perform in $O(n \log n)$. GPU-ABiSort by Greß and Zachmann [4] utilizes *Adaptive Bitonic Sorting* [5], where the key is to use a *bitonic tree*, when merging two bitonic sequences, to rearrange the data to obtain a linear number of comparisons for the merge, instead of the $n \log n$ comparisons required by the standard bitonic sort [6]. This lowers the total complexity of Adaptive Bitonic Sorting to $n \log n$. Greß and Zachmann thereby report slightly faster timings than Govindaraju [2] for their tested 32-bit streams of up to 1M elements.

Recently, Sengupta et al. showed how to efficiently accelerate radix-sort using the GPU and CUDA [7], [8]. According to our measurement, their algorithm is more than 50% faster than the GPU-based bitonic-sort algorithms (see Figure 8).

Mergesort [9] is a well-known sorting algorithm of complexity $O(n \log n)$, and it can easily be implemented on a GPU that supports scattered writing. The GPU-sorting algorithms are highly bandwidth-limited, which is illustrated for instance by the fact that sorting of 8-bit values [10] are nearly four times faster than for 32-bit values [2]. To improve the speed of memory reads, we therefore design a vector-based mergesort, using CUDA and $\log n$ render passes, to work on four 32-bit floats simultaneously, resulting in a nearly 4 times speed improvement compared to merge-sorting on single floats. The Vector-Mergesort of two four-float vectors is achieved by using a custom designed parallel compare-and-swap algorithm, on the 8 input floats to the CUDA shader.

However, the algorithm becomes highly inefficient for the latter m passes, where $m = \log 2p$ and p is the number of processors on the GPU. The reason is that parallelism is lost when the number of remaining lists is fewer than twice the number of processors. We solve this problem by initially using a parallel GPU-based bucketsort [11] dividing the input list into $\geq 2p$ buckets, followed by merge-sorting the content of each bucket, in parallel.

II. Overview of the Algorithm

The core of the algorithm consists of the custom mergesort working on four-float vectors. For each pass, it merges $2L$ sorted lists into L sorted lists. There is one thread per pair of lists to be merged. Initially, there is one list per float4-vector of the input stream. To hide latency and achieve optimal parallelism through thread swapping, the NVIDIA G80-series requires at least two assigned threads per stream-processor. The Geforce 8800 GTX contains 128 stream processors (16 multiprocessors of 8 processor units) and the Geforce 8600 GTS contains 32 stream processors (4 multiprocessors). When enough passes have been executed to make L become less than twice the number of stream processors on the GPU, parallelism is lost and efficiency is heavily reduced, as can be seen in Figure 2. To maintain efficiency, we therefore initially use bucketsort to divide the input stream into at least twice as many lists as there are stream processors, where all elements of list $l+1$ are higher than the elements of list l. The lists should also preferably be of nearly equal lengths! Our custom vector-Mergesort is then executed, in parallel, on each of these internally unsorted lists. A histogram is computed for selecting good pivot-points for the buckets to improve the

probability of getting equal list lengths. This is not necessary for correctness, but improves the parallelism and thereby the speed. The histogram can be recursively refined to guarantee always getting perfect pivot-points. However, we have found just one step to be sufficient in practice for our test cases.

These are the steps of the algorithm, in order:

a) The Histogram for Pivot Points: The pivot points required for the bucketsort algorithm to split the input-list of size N into L independent sublists are created in a single $O(N)$ pass by computing a histogram of the input-lists distribution and using this to find $L-1$ points that divides the distribution into equally sized parts.

b) The Bucketsort Step: We use Bucketsort to split the original list into L sublists that can be independently sorted using the vector-Mergesort algorithm. This pass too is of complexity $O(N)$.

c) The Vector-Mergesort step: is executed in parallel on the L sublists. The elements are grouped into 4-float vectors and a kernel sorts each vector internally. The vector-mergesort then merges each sequential pair of vectors into a sorted array of two vectors (8 floats). In each pass, these arrays are merged two-and-two, and this is repeated in log(L) steps until the entire sublist is sorted. The complexity of the complete Vector-mergesort step is $O(Nlog(L))$

III. Vector-Mergesort

The idea of the classic Mergesort is to split the list into two equal halves, recursively sort the two halves and then merging them back together. Merge-sorting a list bottom up can be done in $(\log n)$ passes with $2^{(\log n)-p}$ parallel merge operations in each pass p, and thus it seems suitable for implementation on a highly parallel architecture such as the GPU.

The programming language CUDA from NVIDIA gives access to some capabilities of the GPU not yet available through the graphics APIs, most notably a per-multiprocessor shared memory and scattered writing to device memory.

Our implementation of the mergesort algorithm works on internally sorted float4 elements instead of on the individual floats, using a novel algorithm. This algorithm has obvious advantages on a vector processor (such as the ATI Radeon HD X2000-series) since it utilizes the vector-parallelism, but is also faster on the scalar G80 architecture as the unavoidable memory latency can be hidden by working on a larger chunk of memory at a time (the G80 can read 128-bit words in a single instruction).

Algorithm

The first step in our algorithm is to internally sort all float4 vectors of the full input array. This can be done very quickly with a small kernel that sorts the elements in three vector operations using bitonic sort [6]. One thread per vector is executed with the following code:

```
sortElements(float4 r)
    r = (r.xyzw > r.yxwz) ? r.yyww : r.xxzz
    r = (r.xyzw > r.zwxy) ? r.zwzw : r.xyxy
    r = (r.xyzw > r.xzyw) ? r.xzzw : r.xyyw
```

In the following passes, the input to each thread is always two sorted vector arrays A and B and the output is the sorted merge of these two arrays. One vector, a, will be taken from A and one vector, b, from B, and the components of these two vectors, a and b, will be sorted so that a contains the lowest four floats and b the highest four floats.

This is easily done if one realizes that, for an element $a[n]$ to be among the four highest elements, there must be at least 4 elements lower than it, and since there will be n lower elements in a that there must be $4-n$ lower elements in b. I.e., $b[4-n-1]$ must be lower than $a[n]$. Thus this can be accomplished by two compare-and-swap instructions, see Figure 1(a).

```
// get the four lowest floats
a.xyzw = (a.xyzw < b.wzyx) ? a.xyzw : b.wzyx
// get the four highest floats
b.xyzw = (b.xyzw >= a.wzyx) ? b.xyzw : a.wzyx
```

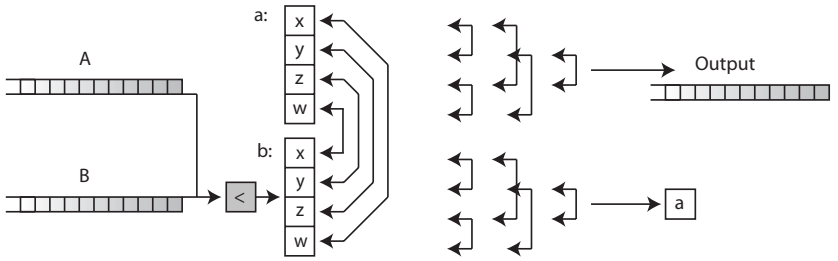
The vectors a and b are then internally sorted by calling `sortElements(a)` and `sortElements(b)`. a is output as the next vector on the output and b takes the place of a (see Figure 1(b)). A new b vector is fetched from A or B depending on which of these contains the lowest value. This process, adding one sorted vector to the output, is repeated until either A or B is empty. When either input array is empty, the (already sorted) remains in the other array are simply appended to the output.

Each thread will be working on some specific part of the input stream, so the arguments passed on to each thread in each pass is the offset where its two consecutive lists begins and the number of elements it shall merge. The thread will start reading its input values from this offset in the input list and will start writing the result into the same offset of the output list. In the next pass, the input and output lists will be swapped.

This method of merge-sorting a list works very well in the first passes, where many threads can merge many lists in parallel. Looking at the execution times for each pass, we can easily see that the algorithm will become slow when there is not enough parallel threads left to keep all processors busy (see Figure 2). In the final step, the entire sorted result will be created by a single processor of a single multiprocessor, on the G80 effectively leaving all but one processor idle. See Figure 3 for pseudo code.

IV. Bucketsort

To increase the parallelism of the mergesort algorithm described above, we need a means of splitting the original list into L sublists where any element in sublist l is smaller than all elements of sublist $l+1$. These sublists can be independently merge-sorted to create the complete sorted array. If $L = 2 * number\_of\_processors$ and the size of each sublist is roughly N/L, where N is the number of elements in the input stream, the mergesort algorithm will be optimally utilizing the hardware, for reasons explained in section 2. I.e., each eight-



(a) Merge-sorting two arrays A and B: In the first iteration of the loop, a and b are taken as the first elements of A and B. In all subsequent iterations, a is the remainder from the previous iteration and b is taken as the element from A or B that contains the lowest value. The components of a and b are then swapped such that a contains the lowest values.

(b) a and b are internally sorted, a is then pushed as the next element on the sorted output and b is the remainder, which will be used in the next iteration.

Fig. 1. The vector-Mergesort kernel

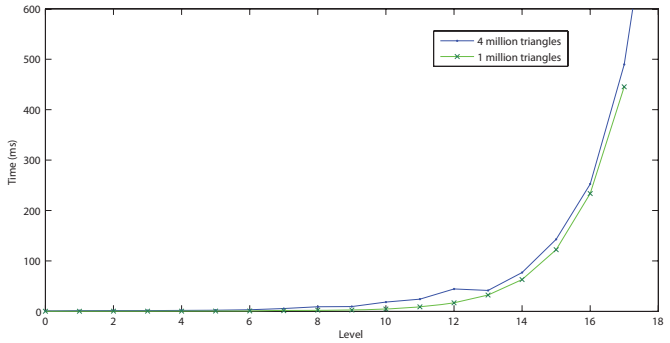


Fig. 2. Time taken for each pass, L, of the mergesort algorithm on a Geforce 8800GTX with 128 cores.

```
mergeSortPass(int nrElements)
    startOfA = threadID * nrElements
    startOfB = startOfA + nrElements/2
    aIdx = 0, bIdx = 0, outIdx = 0
    float4 a = input[startOfA]
    float4 b = input[startOfB]
    while(neither list is empty)
        a = sortElem(getLowest(a,b))
        b = sortElem(getHighest(a,b))
        output[outIdx++] = a
        float4 nextA = input[startOfA + aIdx + 1]
        float4 nextB = input[startOfB + bIdx + 1]
        a = b
        aIdx += (nextA.x < nextB.x) ? 1 : 0;
        bIdx += (nextA.x < nextB.x) ? 0 : 1;
        b = (nextA.x < nextB.x) ? nextA : nextB;
    output[outIdx++] = a;
    while(aIdx < nrElements/2)
        output[outIdx++] = input[startOfFirstList + aIdx++]
    while(bIdx < nrElements/2)
        output[outIdx++] = input[startOfSecondList + bIdx++]
```

Fig. 3. Pseudo code for one CUDA core merging two sorted arrays

core multiprocessor should execute at least sixteen threads for efficient hiding of latency.

To achieve this, we have implemented an algorithm resembling *bucketsort* [11] which uses yet another feature of newer NVIDIA cards (currently 8500 and 8600 but not 8800), namely atomic operations on device memory. The first pass takes a list of $L-1$ suggested pivot points, that divides the list into L parts, and then counts the number of elements that will end up in each bucket, while recording which bucket each element will end up in and what index it will have in that bucket.

The resulting list of the number of elements in each bucket is read back to the CPU and it can be decided if the pivot points were well chosen or if they need to be moved to achieve roughly equally sized lists, as explained below. When the pivot points are changed, the counting pass is repeated.

When the sublist sizes are sufficiently equal, a second pass is run where elements are simply moved to their new positions.

Algorithm

In most cases, little or nothing is known about the distribution of the elements, but the range, or a rough estimate of the range of the elements being sorted is known. If we know nothing about the maximum and minimum of the elements we are sorting, they can be found in a single pass through the elements. On the GPU, all elements could be drawn into a framebuffer and the glMinMax function could be used to find the minimum and maximum elements (at a cost of $\sim 1\%$ of the total execution time). The initial pivot points are then chosen simply as a linear interpolation from the min value to the max value.

Counting Elements per Bucket: The pivot points are first uploaded to each multiprocessor's local memory. One thread is created per element in the input list and each thread then picks one element and finds the appropriate bucket for that element by doing a binary search through the pivot points. Then, the thread executes an *atomicInc* operation on a device-memory counter unique to that bucket, and also stores the old value of that counter, in a separate array at the element's index location. When all elements are processed, the counters will be the number of elements that each bucket will contain, which can be used to find the offset in the output buffer for each sublist, and the saved value will be each elements index into that sublist. See figure 4 for pseudo code.

Refining the pivot points: Unless the original input list is uniformly distributed, the initial guess of pivot points is unlikely to cause a fair division of the list. However, using

```
bucketcount(in inputlist, out indices,
            out sublist, global buckets)
    element = input[threadid]
    index = (#sublists/2) - 1
    jump = #sublists/4
    // Binary search through pivot points to find
    // the bucket for element
    pivot = pivot points[index]
    while(jump >= 1)
        index = (element < pivot)?
            (index - jump):(index + jump)
        pivot = pivot points[index]
        jump /= 2
    index = (element < pivot)?index:index+1 // Bucket index
    sublist[threadid] = index //Store the element
    // Increase bucket size and store it as index in bucket
    indices[threadid] = atomicInc(buckets[index])
```

Fig. 4. Pseudo code for counting the number of elements per bucket

the result of the first count pass and making the assumption that all elements that contributed to one bucket are uniformly distributed over the range of that bucket, we can easily refine the guess for pivot points, as shown in Figure 5.

```
Start with first pivot point
bucket[n] is the number of elements in bucket n
elemsneeded = N/L //#elemets wanted in each bucket
// Try to take as many elements as needed, assuming that the
// distribution is even within the bucket we are taking from
for each n in buckets
    range = range of this bucket
    while(bucket[n] >= elemsneeded)
        next pivotpoint += (elemsneeded/bucket[n])*range
        output one pivotpoint
        elemsneeded = N/L
        bucket[n] -= elemsneeded
    elemsneeded -= bucket[n]
    next pivotpoint += range
```

Fig. 5. Refining the pivots

Running the bucketsort pass again, with these new pivot points, will result in a better distribution of elements over buckets. If these pivot points still do not divide the list fairly, the pivot points can be further refined in the same way. However, a single refinement of pivot points was sufficient in practice for our test cases.

Repositioning the elements: When a suitable set of pivot points are found, the elements of the list can be moved to their new positions as recorded in the counting pass. A prefix sum is calculated over the bucket sizes so that an offset is obtained for each bucket, and each element is written to its buckets offset plus its recorded index (see Figure 6 for pseudo code). In this way, the bucketsort requires a minimum amount of storage.

```
bucketsort(in inputlist, in indices, in sublist
           out outputlist, global bucketoffsets)
    newpos = bucketoffsets[sublist[threadid]]
             + indices[threadid]
    outputlist[newpos] = inputlist[threadid]
```

Fig. 6. Moving the elements to their buckets

Optimizations: Since we can be quite sure that the initial guess of pivot points will not be good enough, and since the initial guess is simply a linear interpolation from the minimum to the maximum, the first pass can be very much optimized by realizing that all we really do is to create a histogram of the distribution. Thus, in the first pass, we do not store the bucket or bucket-index for the elements, and we do not need to do a binary search to find the bucket to increase (see Figure 7). The steps of the bucketsort then becomes:

```
histogram(in inputlist, in min, in max, global buckets)
    element = input[threadid]
    index = ((element - min)/(max-min)) * L
    atomicInc(buckets[index])
```

Fig. 7. Pseudo code for creating the histogram using CUDA

- Creating Histogram
- Refining Pivots
- Counting elements per bucket
- Repositioning the elements

When creating the offsets for each bucket for the final pass, we also take the opportunity to make sure that each bucket starts at a float4 aligned offset, thereby eliminating the need for an extra pass before merge-sorting the lists.

Choice of divisions: As explained above, we will need to split the list into at least $d = 2p$ parts, where p is the number of available processors. It is easy to realize, though, that since each thread will do an atomic increase on one of d addresses, that choice would lead to a lot of stalls. However, the mergesort algorithm does not impose any upper limits on the number of sublists. In fact, increasing the number of sublists in bucketsort decreases the amount of work that mergesort needs to do. Meanwhile, splitting the list into too many parts would lead to longer binary searches for each bucketsort-thread and more traffic between the CPU and GPU. In our tests, with 32 processors (on the GeForce 8600) splitting the list into 1024 sublists seems to be the best choice.

In the histogram pass, where no binary search is required, we could however use several times more buckets to improve the quality of pivot points if the distribution is expected to be very uneven and spiky.

Random shuffling of input elements: Since the parallel bucketsort relies on atomic increments of each bucket size, nearly sorted lists can cause many serialized accesses to the same counter, which can limit the parallelism and lower the speed. In order to avoid significant slow-down we can therefore add one pass, without compromising generality, at the beginning of the algorithm, which randomly shuffles the elements to be sorted. This is done in $O(N)$ time using CUDA and the atomic exchange instruction.

V. Complete Algorithm

The mergesort algorithm presented above is extremely fast until the point where merging lists can no longer be done in parallel. To achieve high parallelism, we start out using bucketsort to create L sublists where any element in sublist l is smaller than any element in sublist $l + 1$. Merge-sorting

each of these sublists can be done in parallel and the result is the complete sorted list.

Mergesorting lists of arbitrary sizes: The mergesort algorithm described above requires the input list size to be a multiple of four elements while the bucketsort algorithm can create sublists of any size. Mergesorting lists of arbitrary sizes is solved by initializing the bucketsort output with values lower than the minimum, and float4 aligning the bucket offsets. Then, when the sublists are sorted, the invalid numbers will be the first in each sublist and can trivially be removed from the result.

Our CUDA implementation of the mergesort algorithm works such that the *y-index* of the block gives its *sublist* index while the block *x-index* and the thread *x-index* together give the part of the sublist being worked with. A local memory int-array gives the number of elements in each sublist and it is up to each thread to make sure that they are not working on elements outside of the actual sublist, and if they are, exit.

For load balancing, it is important to keep the sizes of the sublists, which are the result of the bucketsort, to about the same size.

VI. Result

Our algorithm was tested on a Pentium D system at 2.8 GHz with a GeForce 8600GTS graphics card. GPU-based radix-sort [7] and the bitonic sort based GPUSort algorithm [2] were run on the same machine, and the results are shown in Figure 8. For the comparison-algorithms, we used the authors' implementations. For practical reasons, the entire single CPU quicksort times are not fully shown in the graph, but 1M elements takes 0.561s and 8M elements takes 8.6s. This is 6-14 times slower than our GPU-based hybrid algorithm for 1-8M elements respectively.

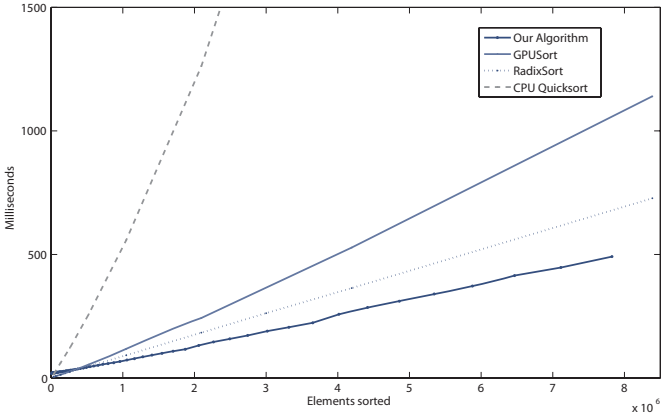


Fig. 8. Time taken to sort arrays of varying sizes with our algorithm, radix-sort [7], GPUSort [2], and a CPU-Quicksort. A random distribution in [0, #elements] was used.

Gress and Zachmann [4] present results of up to 1M elements, and then report a 37% performance improvement on the GPUSort algorithm on a GeForce 6800 based system, but only about 5% on a Geforce 7800 system. Our algorithm performs more than twice as fast as the GPUSort algorithm for arrays of four million elements and more. Also, our algorithm handles arrays of arbitrary sizes while these two bitonic sort based algorithms require list-sizes to be a power of two. The bitonic sort algorithms and the radix-sort are, however, faster for sizes below 512k elements, and their speed is independent on the input distribution.

In the Fig. 8, only the time to actually sort the elements was measured for the radix-sort, GPUSort and our algorithm. The reason is that the upload/download-functions in CUDA are faster than the corresponding OpenGL-functions, which would give us an unfair advantage in the comparisons. The time taken to upload and download the elements to and from the GPU on our system using CUDA constitutes about 10% of the total time, as can be seen in Fig. 9.

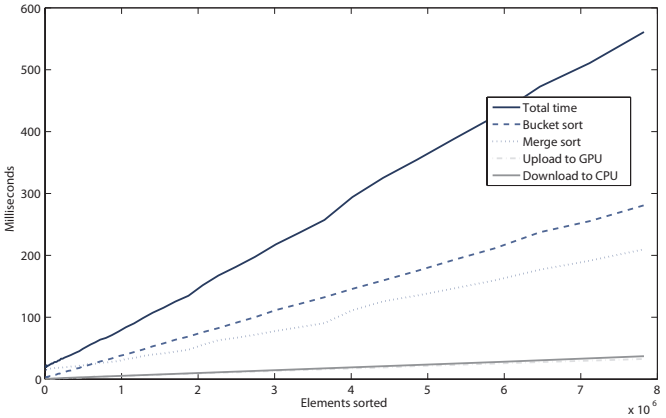


Fig. 9. Time taken by the different parts of our algorithm. The same input as in Fig. 8 was used.

Distance Sorting

We here demonstrate the algorithm for distance sorting on two scenes. Distance sorting is for example a common operation in computer graphics for particle systems and achieving correct transparency. Vertices in vertex-lists have a tendency to appear in a fairly non-random order and therefore we use the option of randomly shuffling the input elements. This shuffling step takes an approximate 10% of the total execution time. As seen in Figure 10, our algorithm performs the sorting faster than radix-sort and bitonic-sort for the two test scenes.

VII. Conclusions

We have presented a GPU-based sorting algorithm. It is a hybrid that initially uses one pass of bucket-sort to split the input list into sublists which then are sorted in parallel using a vectorized version of parallel merge-sort. For input arrays of more than 512K elements, we show that the algorithm performs significantly faster than prior GPU-based sorting algorithms, and it is an order of magnitude faster than single CPU quicksort. The execution time is, however, not insensitive to the input distribution. To ameliorate this, a histogram is used to improve load-balancing, and optionally, an initial random shuffling of the input elements could be used to avoid serialization effects in the bucket-sorting.

REFERENCES

[1] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for gpus," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*. New York, NY, USA: ACM Press, 2006, p. 184.

[2] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: High performance graphics coprocessor sorting for large database management," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2006.

[3] P. Kipfer and R. Westermann, "Improved gpu sorting," in *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, 2005, pp. 733–746.

[4] A. Greß and G. Zachmann, "Gpu-abisort: Optimal parallel sorting on stream architectures," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, April 2006, p. p 45.

[5] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines," *SIAM J. Comput.*, vol. 18, no. 2, pp. 216–228, 1989.

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

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

[8] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *GPUGems 3*, 2007, pp. 851–876.

[9] D. Knuth, "Section 5.2.4: Sorting by merging," in *The Art of Computer Programming, Volume 3 - Sorting and Searching*. ISBN 0-201-89685-0, 1998, pp. 158–168.

[10] N. K. Govindaraju, N. Raghuvanshi, M. Henson, D. Tuft, and D. Manocha, "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," in *UNC Technical Report*, 2005.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Section 8.4: Bucket sort," in *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, ISBN 0-262-03293-7, 2001, pp. 174–177.

(a) Test scene: 16 robots in a jungle. 1.54M triangles.

(b) Thai statue. 5M triangles.

Scene	Our	OH	w/o rnd	Radix	GPUSort_i	GPUSort_u	QSort
(a)	123	10.1	336	135	170	250	1150
(b)	363	32	343	434	641	1140	4775

Fig. 10. Timings in milliseconds for sorting of distance values for three scenes. The first scene is a jungle scene with 16 robots, consisting of 1.54M triangles in total. The second scene is the Stanford Thai-statue of 5M triangles. *Our* is total execution time for our algorithm. *OH* is the overhead caused by the shuffling step (included in *Our time*). *W/o rnd* is the time for our algorithm without doing the random shuffling step. *Radix* shows the time for GPU-based radix-sort [7]. GPUSort requires lists sizes to be a power of two and therefore GPUSort_i shows virtual, interpolated timings as if GPUSort could handle arbitrary sizes, while GPUSort_u lists real timings for the closest upper power of two size. *QSort* is the same CPU-based quicksort used in [2].