Fast Parallel GPU-Sorting Using a Hybrid Algorithm

Erik Sintorn, Ulf Assarsson

Department of Computer Science and Engineering

Chalmers University Of Technology

Gothenburg, Sweden

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, GPU-based bucketsort or quicksort splits the list into enough sublists then to be sorted in parallel using merge-sort. The algorithm is of complexity $n \log n$, and for lists of 8M elements and using a single Geforce 8800GTS-512, it is 2.5 times as fast as the bitonic sort algorithms, with standard complexity of $n(\log n)^2$, which for long was considered to be the fastest for GPU sorting. It is 6 times faster than single CPU quicksort, and 10% faster than the recent GPU-based radix sort. Finally, the algorithm is further parallelized to utilize two graphics cards, resulting in yet another 1.8 times speedup.

Key words: parallelism, sorting, GPU-algorithms

PACS: 07.05.Hd, 07.05.Kf

1 Introduction

Sorting is a general problem in computer science. Mergesort [11] 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 bitonic sorting of 8-bit values [5] are nearly four times faster than for 32-bit values [4]. 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 each thread running the CUDA core.

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 [3] or quicksort [9,16,18], dividing the input list into $\geq 2p$ buckets, followed by merge-sorting the content of each bucket, in parallel.

The parallel bucketsort, implemented in NVIDIA's CUDA, utilizes the synchronization mechanisms, such as atomic increment, that is available on modern GPUs (e.g. the G92-series consisting of NVIDIA 8500, 8600, and 8800GTS-512). Quicksort instead uses the geometry shaders, allowing execution on hardware lacking atomic operations (e.g. current ATI-cards) but is generally

slightly slower. The mergesort requires scattered writing, which is exposed by CUDA and ATI's Data Parallel Virtual Machine[14].

Finally, the algorithm is adapted to use dual graphics cards, to achieve yet another 1.8 times speedup. As a realistic test case, we demonstrate the performance on sorting vertex distances for two large 3D-models; a key in for instance achieving correct transparency.

2 Previous Work

Since sorting is a vastly researched area, we here present only the most related previous work.

Bitonic sort has primarily been used by previous GPU sorting algorithms even though the classic complexity is of $n(\log n)^2$ [4,10]. It is, however, possible to modify bitonic sort to perform in $O(n \log n)$. GPU-ABiSort by Greß and Zachmann [6] utilizes *Adaptive Bitonic Sorting* [2], 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 [1]. This lowers the total complexity of Adaptive Bitonic Sorting to $n \log n$. Greß and Zachmann thereby report slightly faster timings than Govindaraju [4] 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 [16,8]. The complexity of radix-sort is also $n \log n$. According to our measurement, their algorithm is roughly 50% faster than the GPU-based bitonic-sort algorithms (see Figure 7).

GPU-based quicksort has been presented by both Sengupta et al. [16] and Sintorn and Assarsson [18]. The former uses CUDA and the latter uses the geometry shader. Neither solution is fast for fully sorting the input list. However, quicksort can successfully be used to quickly achieve a partial sort. We select the solution of Sintorn and Assarsson, since that source code was available to us. More details are given in Section 5.2.

3 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 and 8800GTS-512 contain 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 either bucketsort or quicksort 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 bucketsort 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. In theory, we could use the histogram for generating good pivot-elements for quicksort as well. However, our current implementation of the histogram requires the atomicInc-operation, and if this is available, bucketsort is a faster choice.

These are the steps of the algorithm, in order:

(1) *Partitioning into L sublists* using either
   (a) *quicksort*, of complexity $O(Nlog(L))$, *or*
   (b) *bucketsort*. The pivot points to partition 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. The bucketsort then relocates the elements into their buckets. This pass too is of complexity $O(N)$.

(2) *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 following 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 $O(Nlog(L))$

4 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 $\frac{n}{2^p}$ 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 [1]. In this section, we will use a compact notation to describe vector component swizzling, similar to that used in shader languages. For example, `a.xyzw = b.wzyx` means that the vector a will take on the values of vector b in reversed order. One thread per

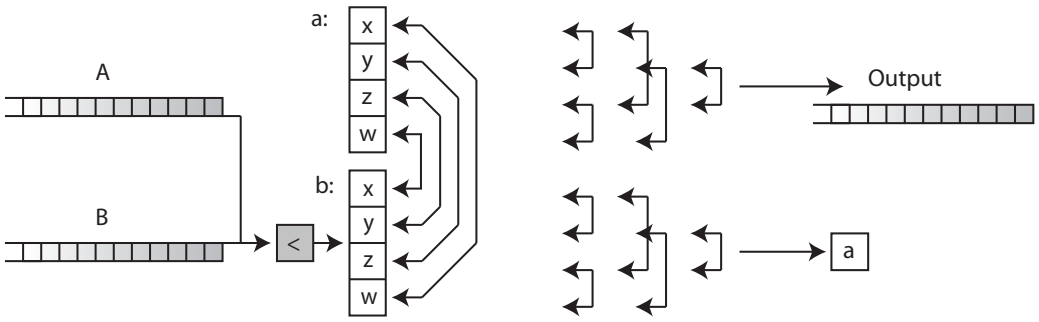
vector is executed with the following code:

```
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
```



(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. The double-headed arrows in the figure denote a compare-and-swap operation.

Fig. 1. The vector-Mergesort kernel

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.

5 Partitioning of input stream into sublists

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

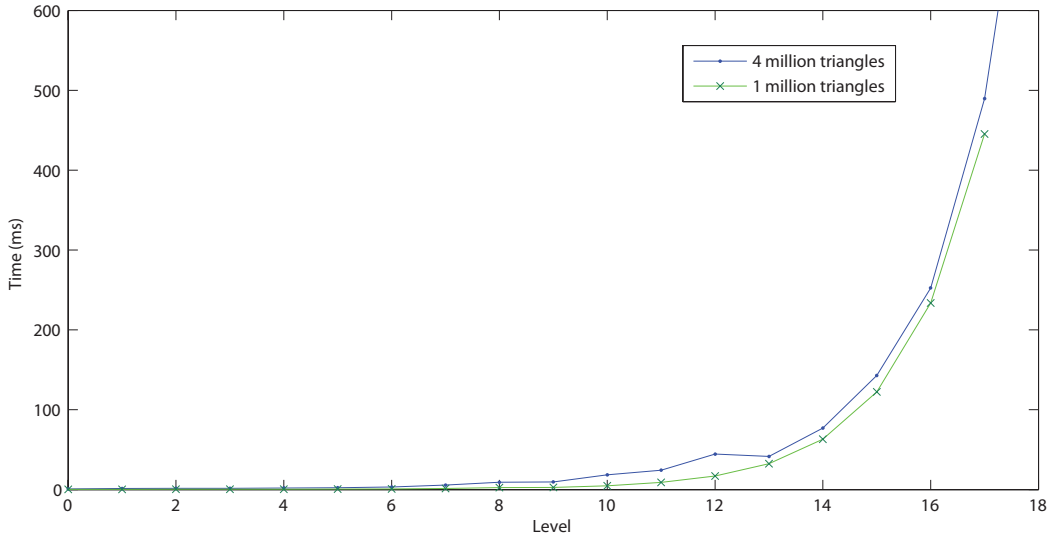


Fig. 2. Time taken for each pass, L, of the mergesort algorithm on a Geforce 8800GTX with 128 cores.

```
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[startOfA + aIdx++]
    }
    while(bIdx < nrElements/2) {
        output[outIdx++] = input[startOfB + bIdx++]
    }
}
```

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

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

To achieve this, we have implemented one step of *bucketsorting* [3], and as an alternative, a partial quicksort [9,18]. The bucketsort is faster, but requires an atomicInc, which is not available on all graphics cards.

5.1 Bucketsort

We have implemented an algorithm resembling *bucketsort* which uses yet another feature of newer NVIDIA cards, 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 sweep through the elements. On the GPU, it is more efficient to perform a reduction [7], with a cost of $< 1\%$ 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 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

```
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. atomicInc increases the specified global-memory location by one and returns the previous value in one indivisable instruction.

pivot points, as shown in Figure 5.

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

```
Start with first pivot point
bucket[n] is the number of elements in bucket n
N = total number of elements
L = number of buckets
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

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. We modified NVIDIA's 256 bucket histogram [15] to work with up to 1024 buckets. This modification is straight forward. However, the limited size of the fast shared memory forces us to use 4 times fewer threads,

leading to slightly less processor utilization. Still, this pass only takes 4 ms of totally 140 ms for 8M elements. The steps of the bucketsort then becomes:

- 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, splitting the list into 1024 sublists seems to be the best choice, both when using 32 processors (on the GeForce 8600) and using 128 processors (on the GeForce 8800GTS-512).

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.

5.2 Quicksort

As an alternative to bucketsort, a few passes of quicksort can be run to divide the input stream into L sublists. The advantage is that no atomic synchronization instructions are needed. The drawback is that it is slightly slower and tends to create lists of larger variance in lengths, since the histogram is not used to select good pivots. The histogram can be used, but currently requires the atomicInc, which violates the purpose of choosing quicksort for dividing into sublists. The algorithm works as described by Sintorn and Assarsson [18], but here follows a recap.

The *geometry shader* receives the input list as a stream of points. In the first pass, all elements higher than the pivot are discarded, and the rest is kept by using the *Transform Feedback* extensions of OpenGL [13]. The number of elements in the resulting list can be found with a query. The second pass is identical but instead keeping all elements lower or equal than the pivot. The process is then repeated recursively on the two resulting sublists.

If several recursive steps are used, the overhead of managing many sublists

becomes dominating. However, when only recursing a few steps, this algorithm is still efficient. The result is a split of the input list into L sublists, where each element of sublist $l + 1$ is guaranteed to be higher than those of sublist l, which exactly corresponds to the required input to the mergesort.

6 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 or quicksort to create L sublists where any element in sublist l is smaller than any element in sublist $l + 1$. Mergesorting 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 our bucketsort and quickort can create sublists of any size. Mergesorting lists of arbitrary sizes is solved by float4 aligning the sublist offsets and initialize unused elements with values lower than the minimum. 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 desirable that the generated sublists have about the same size. For bucketsort, this is the purpose of using the histogram.

6.1 Parallelizing using dual graphics cards

If there are available slots on the motherboard, more than one graphics card can be used. We tested our hybrid of bucketsort and mergesort on dual graphics cards. The parallelization is performed by executing the histogram generation on both graphics cards (see Section 5.1). From this histogram, it is easy to estimate a pivot that divides the input elements in two nearly equally sized halves of elements on each side of the pivot. The first graphics card handles the elements lower than and equal to the pivot and the second card the higher elements. In the same way, it would be trivial to parallelize for more than two cards, as long as the first histogram pass corresponds to a small amount of the total execution time. Currently, it takes 4 ms out of totally 140 ms for sorting 8M elements on a single card. This means that a parallel sort using n cards would require $\frac{150}{n} + 4$ ms, theoretically providing good parallelization even for dozens of cards.

7 Result

Our algorithm was tested on a Intel Core2 Duo system at 2.66 GHz with a single GeForce 8800GTS-512 graphics card and also with dual cards. GPU-based radix-sort [16] and the bitonic sort based GPUSort algorithm [4] were run on

the same machine using a single graphics card, since these algorithms currently are not adapted for more than one card. The results are shown in Figure 7. Here, our hybrid sort uses the bucketsort followed by mergesort, since using partial bucketsort was found to be $\sim 45\%$ faster than using partial quicksort for 1-8M elements. For the comparison-algorithms, we used the authors' implementations. The CPU algorithm is the STL-Introsort implementation [12] found in the STL-library for Microsoft Developer Studio 2005, that runs on a single core. It is based on quicksort, but switches to heap-sort for small enough sublists. For practical reasons, the entire single CPU quicksort times are not fully shown in the graph, but 8M elements takes 1.26s. This is 6.2 times slower than our single GPU algorithm and 11 times slower than the dual GPU version. In all of our experiments, time measurements are done several times and the average running time is presented.

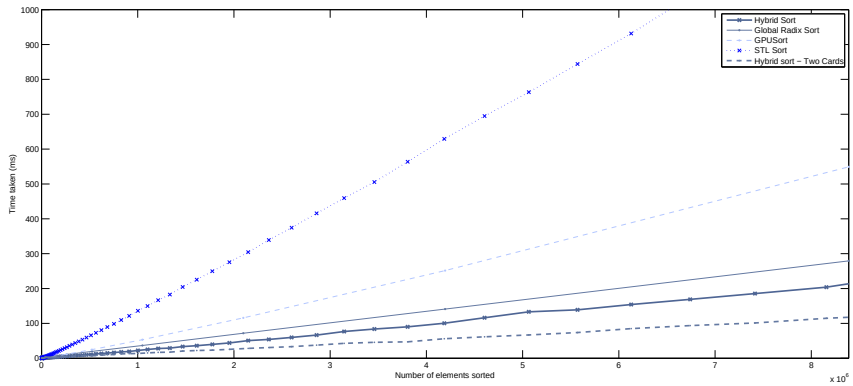


Fig. 7. Time taken to sort arrays of varying sizes with our algorithm, radix-sort [16], GPUSort [4], and a CPU-Quicksort. Finally, we also test our hybrid method using dual graphics cards. A random distribution in [0, #elements] was used.

Gress and Zachmann [6] 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 algo-

rithm performs more than twice as fast as the GPUSort algorithm. 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 speed for the bitonic sort and radix sort is, however, independent on the input distribution.

In the Fig. 7, only the time to actually sort the elements was measured for the radix-sort, GPUSort and our hybrid 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. 8.

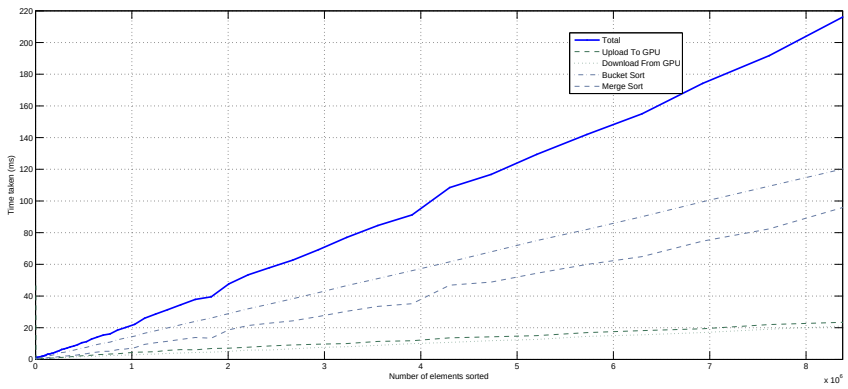


Fig. 8. Time taken by the different parts of our algorithm. The same input as in Fig. 7 was used.

Distance Sorting

As a realistic test scenario, 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. The results are shown in Figure 9. It should be noted that the main reason we do not achieve ideal speedup with dual cards is that the shuffling step has to be performed before constructing the histogram. Thus, the shuffling must be executed either on each card for all elements, or in parallel for half of the elements on each card and then communicating the results to each card. For the latter option, CUDA currently requires the communication to go via the CPU, which is too slow to be worthwhile. Hopefully, future versions will allow communication directly over the SLI-bridge or PCI-port.

Our algorithm, as most GPU based sorting algorithms, is memory-latency bound throughout. In the bucketsort pass, the atomicInc operations are the main bottleneck, and in the mergesort pass the uncoalesced global memory read and write operations are the bottleneck.

8 Conclusions and Discussion

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. We show that the algorithm generally performs slightly faster than the radix sort, using a single graphics card, and is significantly faster than other prior GPU-based sorting algorithms. It is an order of magnitude faster than single CPU quick-sort. Our algorithm is trivial to further parallelize using dual graphics cards, resulting in additionally 1.8 times speedup. The execution time is, however,

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

(b) Thai statue. 5M triangles.

Scene	Our dual	Our single	OH	w/o rnd	Radix	GPUSort$_i$	GPUSort$_u$	STL Sort
(a)	41	61	5.4	106	54	83	116	216
(b)	93	160	19	167	169	309	549	755

Fig. 9. 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 dual* is total execution time for our method, using two graphics cards and including shuffling. For all the other figures, only one graphics card is used. *Our single* is total execution time for our algorithm with a single card. *OH* is the overhead caused by the shuffling step (included in *Our single*). *W/o rnd* is the time for our algorithm without doing the random shuffling step. *Radix* shows the time for GPU-based radix-sort [16]. 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 (for comparison reasons only), while GPUSort$_u$ lists real timings for the closest upper power of two size. *STL Sort* is a CPU-based quick sort implementation.

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.

It would of course be trivial to use the bucketsort to split the input stream into a nearly equally sized upper and lower half, and then sort each half on one graphics card using radix sort. However, little would be gained, since the bucketsort is the input-sensitive part of the algorithm and our algorithm in all our test cases performs similar or better than radix sort. Especially on the Geforce 8600 GTS, with only 32 stream processors, where it performs up to 40% faster [17].

References

[1] K. E. Batcher, Sorting networks and their applications, Proc. AFIPS Spring Joint Computer Conference 32 (1968) 307–314.

[2] G. Bilardi, A. Nicolau, Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines, SIAM J. Comput. 18 (2) (1989) 216–228.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, 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.

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

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

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

[7] M. Harris, Optimizing parallel reduction in cuda (2007).
URL
http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/

[8] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with cuda, in: GPUGems 3, 2007.

[9] C. A. R. Hoare, Partition: Algorithm 63, quicksort: Algorithm 64, and find: Algorithm 65., Comm. of ACM 4 (7) (1961) 321–322.

[10] P. Kipfer, R. Westermann, Improved gpu sorting, in: M. Pharr (ed.), GPU Gems 2, Addison Wesley, 2005, pp. 733–746.

[11] 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.

[12] D. R. Musser, Introspective sorting and selection algorithms, Software - Practice and Experience 27 (8) (1997) 983–993.
URL citeseer.ist.psu.edu/musser97introspective.html

[13] NVIDIA, Nv_transform_feedback opengl extension (2006).
URL
http://www.opengl.org/registry/specs/NV/transform_feedback.txt

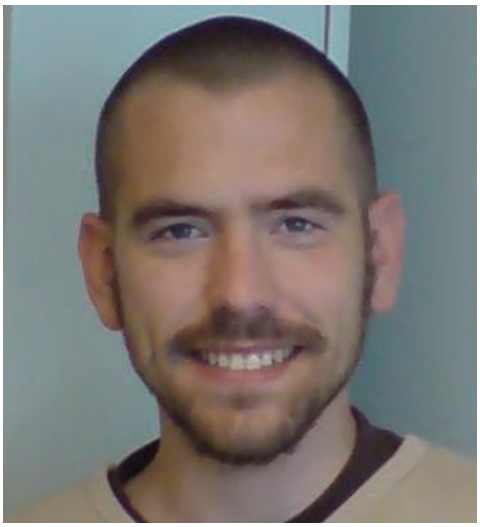
[14] M. Peercy, M. Segal, D. Gerstmann, A performance-oriented data parallel virtual machine for gpus, in: SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches, ACM Press, New York, NY, USA, 2006.

[15] V. Podlozhnyuk, Histogram calculation in cuda (2007).
URL
http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/

[16] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, Scan primitives for gpu computing, in: Graphics Hardware 2007, ACM, 2007.

[17] E. Sintorn, U. Assarsson, Fast parallel gpu-sorting using a hybrid algorithm, Workshop on General Purpose Processing on Graphics Processing Units (GPGPU).

[18] E. Sintorn, U. Assarsson, Real-time sorting for self shadowing and transparency in hair rendering, in: Proceedings of i3D 2008, Symposium on Interactive 3D Graphics and Games, ACM, 2008.



(a) Erik Sintorn received a M.Sc. in 2007 and started his Ph.D. studies the same year at the Department of Computer Engineering at Chalmers University of Technology, Sweden. His main focus is shadows and GPU-algorithms.

(b) Ulf Assarsson received a PhD in Computer Graphics in 2003 and a M.Sc. degree in engineering physics in 1997, both at Chalmers University of Technology. He is now an assistant professor at the Department of Computer Engineering. His research interests include realistic real-time rendering, ray tracing, and GPU- algorithms.