Universität Bremen

Fachbereich 3: Mathematik und Informatik

Diploma Thesis

A Time-Based Adaptive Hybrid Sorting Algorithm on CPU and GPU with Application to Collision Detection

Robin Tenhagen

Matriculation No.2288410

5 January 2015

Examiner: Prof. Dr. Gabriel Zachmann
Supervisor: Prof. Dr. Frieder Nake
Advisor: David Mainzer

Robin Tenhagen

A Time-Based Adaptive Hybrid Sorting Algorithm on CPU and GPU with Application to Collision Detection

Diploma Thesis, Fachbereich 3: Mathematik und Informatik

Universität Bremen, January 2015

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 5. Januar 2015

Robin Tenhagen

Abstract

Real world data is often being sorted in a repetitive way. In many applications, only small parts change within a small time frame. This thesis presents the novel approach of time-based adaptive sorting. The proposed algorithm is hybrid; it uses advantages of existing adaptive sorting algorithms. This means, that the algorithm cannot only be used on CPU, but, introducing new fast adaptive GPU algorithms, it delivers a usable adaptive sorting algorithm for GPGPU. As one of practical examples, for collision detection in well-known animation scenes with deformable cloths, especially on CPU the presented algorithm turned out to be faster than the adaptive hybrid algorithm *Timsort*. For a different cloth animation, the algorithm was faster than Thrust's *Merge Sort* on older graphics hardware.

The thesis investigates different measures of unsortedness, different adaptive CPU sorting algorithms and GPU sorting algorithms in general. Furthermore, it delivers an outline on improvements and the evolution of the new sorting algorithm.

Contents

Chapter 1

Introduction

Sorting is one of the major problems in computer science. Many different algorithms have been developed, including comparison-based, in-place and stable sorts, for example [PSL12]. One aspect is, that there is no generally valid algorithm. Different scenarios require different algorithms for sorting. This thesis has the goal of writing a practical sorting algorithm. "Practical" means "applicable for real world data". In order to understand, what "real world" data means, it is necessary to survey [SW11] the most common practical applications of sorting.

One field is *Commercial Computing*, where databases with address data, or — in general — strings have to be sorted. Various *String Processing* applications, like finding patterns, require sorting algorithms. In general, sorted data is easier to be *search*ed through (for example by humans or also the fast *Binary Search*). Another field is *Operations Research*, where scheduling tasks need to order execution times of processes to organize load-balancing. In the scientific world, *Event-Driven Simulations* require appropriate algorithms, just as in particle- or body-collision simulations. In other applications, *Priority Queues* come into account; some *Numerical Algorithms* require them in order to control accuracy in calculations, *Combinatiorial Searches*, like the $A*$ algorithm, use them in areas of artificial intelligence and *Graph Searches* like *Prim's Algorithm* and *Dijkstra's Algorithm* utilize them as well. Also *Huffman Compression* uses *Priority Queues* and in *Kruskal's Algorithm*, sorting plays a vital role for ordering edge weights in graphs. All these applications make efficient sorting algorithms indispensable. One objective for this thesis is to analyze practical data and use this knowledge in order to create an adaptive, "more realistic" sorting algorithm. An *Adaptive Sorting Algorithm* is a sorting algorithm that is more efficient, faster, for pre-sorted data. This means, that based on the data analysis, an appropriate algorithm needs to be designed. Since the previously mentioned list of sorting applications has a wide range, this work will only focus on one application: Collision detection between rigid and deformable models, specifically cloth simulations.

Collision detection methods need "to check if collisions occur between a pair of objects as well as self-collisions among deformable objects. In many applications, an additional requirement is that the collision detection has to be calculated within milliseconds." [MZ14]. Often, physical

simulations or video games have this requirement. To reduce computation time, improvements such as *Spatial Subdivision* or *Approximation of Surfaces* have evolved. These algorithms employ *Axis-Aligned Bounding Boxes* (AABB) [Ber97], *Oriented Bounding Boxes* (OBB) [GLM96] or *Inner Sphere Trees* (IST) [WZ09]. In order to reduce pairs of primitives that need to be checked for collision, *Culling Methods* such as *Sort and Sweep* [Bar92] were introduced. A common approach to speed up collision detection of rigid and deformable objects involves *Bounding Volume Hierarchies* (BVH) [Eri05]. Other improvements employed *Precomputed Chromatic Decomposition of a Mesh* [Gov+05b] or *Stenciled Geometry Images* [GGK06] to create GPU-optimized BVHs. The *Hybrid CPU-GPU Parallel Continuous Collision Detection* (HPCCD) [Kim+09] was also based on BVHs. Nevertheless, this hybrid solution involved a huge communication traffic between GPU and CPU for reconstructing BVHs on the CPU [MZ14]. A pure GPU-based linear BVH approach can be found in [LMM10] and another GPU-based streaming algorithm for collision detection between deformable objects can be found in [Tan+11].

This thesis focuses on a simple method of collision detection, it is based on *Axis-Aligned Bounding Boxes*. This simple approach without the introduction of additional complex data structures has the advantage, that it works on the whole scene and can be easily implemented. In fact, it is possible to port it easily to GPU streaming processors without having to deal with complex memory management. As a further simplification, the base collision detection will merely take place in one dimension; bounding boxes will be projected on the X-axis only.

The structure of the thesis is as follows: First, previous work on sorting algorithms (chapter 2.4) will be presented and it will be explained, what pre-sortedness actually is (chapter 2.1). Afterwards, commonly available cloth simulations will be chosen (chapter 4.1) and their projected bounding box array will be examined to identify pre-sorted data (chapter 5.1). Later, the evolution of the consequently developed new algorithm will be presented in chapter 6. Detailed performance results complete the analysis (chapter 7).

Chapter 2

Related Work

2.1 Unsortedness

2.1.1 Measures of Unsortedness

In order to understand what it means to take advantage of pre-sortedness, it has to be defined more closely what unsortedness is. Ottmann and Widmayer give an outline to three measures of describing unsortedness (see [OW12]). For better understanding, the following sequence of sort keys shall be investigated:

$$S_a = 9\ 1\ 2\ 3\ 5\ 4\ 7\ 6$$

2.1.1.1 Inv

This measure is a way of describing unsorted pairs, the number of inversions. The second half of S_a, for example, has intuitively the unsorted pairs of (5,4) and (7,5). Although as for the example this measure seems to look at local positions, it is actually a way to express global unsortedness, as for the 9 will have to be swapped all the way through the data: (9,1), (9,2), (9,3), (9,5), (9,4), (9,7), (9,6) are the unsorted pairs here. This leads to an *inv* measure of 9 elements for S_a. A formal way to describe this measure is

$$inv(S) = |\{(i,j)|1 \leq i < j \leq n \text{ and } k_i > k_j\}|$$

The minimum value of unsorted pairs for a sorted array is 0. The maximum value for an inversely sorted array is

$$inv(S_{inverse}) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

2.1.1.2 Runs

Looking at the left half of S_a, it should be easy to merge already sorted subsequences into one globally sorted sequence. As a matter of fact, (9,1) and (2,3) are mergeable, for example by a bitonic merge as the bitonic sequence (9,1,2,3) (for more information see section 2.5.1). The Runs measure looks exactly at the length of unsorted sequences.

$$runs(S) = |\{i | 1 \leq i \leq n \text{ and } k_{i+1} < k_i\}| + 1$$

The sequence (9,1) is unsorted, (5,4) and (7,6) are also. This makes $runs(S_a) = 3 + 1 = 4$. In case of a sorted array, the number of runs is 1. In an inversely sorted array the number of runs is obviously as big as the number of elements in the array. In general, the Runs measure rather refers to a local degree of unsortedness, seeing the right half of S_a yieldung most unsortedness in this measure.

2.1.1.3 Rem

The position of key 9 produced most unsortedness for the *inv* measure, though. Intuitively, it should not be too complex to move the 9 further, if the elements (1,2,3) are already sorted. The longest ascending subsequence measure (las) considers exactly these sorted subsequences (which do not have to be coherent):

$$las(S) = \max\{t | \exists i(1), \ldots, i(t) \text{ that } 1 \leq i(1) < \ldots < i(t) \leq n \text{ and } k_{i(1)} < \ldots < k_{i(t)}\}$$

For a sorted sequence S, $las(S)$ has the value n and for an inversely sorted sequence S_i, $las(S_i)$ has the value 1. Comparing this to the previous two measures, *las* grows inversely to how the other measures grow. In order to make the measures more comparable, the measure *rem* will be introduced, which considers the number of elements that need to be removed for producing a

sorted sequence. It is defined by

$$rem(S) = n - las(S)$$

For our example, $rem(S_a)$ is $8 - 5 = 3$, with the generated longest ascending subsequence of (1,2,3,4,6). In contrast to the measures inv and las, rem is both, a local and a global way to determine unsortedness. Since the sorted subsequences do not have to be coherent, the algorithm for finding longest ascending subsequences (also called longest increasing subsequences) is less intuitive and its algorithmic problem is not trivial (for more information see [Sch61] [Man85]).

2.1.1.4 Other Measures

A simpler type of measure is max [EW92]. It is defined by the maximum distance any of the items has to overcome to be sorted. This measure has a purely global perspective and even one unsorted item is enough to produce a high level of unsortedness in this measure. For its global and simple perspective, this measure will be used as a first approach for determining sort ranges in the algorithm presented in this thesis, which shall be called *AdaptiveFrameSort*.

Another measure is dis; this measure is "defined by the largest distance determined by an inversion" [EW92]. In the example case, $dis(S_a) = \text{distance}(9, 6) = 7$. There are several other measures, which often are variations of the previously named measures or they are rather based on a theoretical point of view. Since the measure classes of local, global and mixed perspectives are already covered by the previously named measures, for this work it is not important to outline other measures in detail. According to [EW92] these are: exc (minimum number of exchanges required to sort a sequence [Man85]), SUS (a natural version of the $runs$ measure: Shuffled Up-Sequences [CLP93]), $SMS.SUS$ (a further generalization as Shuffled Monotone Subsequence), enc (where $enc(X)$ is "defined as the number of sorted lists constructed by Melsort when applied to X" [EW92], a refined measure based on existing algorithms' analysis, further reading: [Ski88]), osc (which "evaluates [..] the 'oscilation' of large and small elements in a given sequence" [EW92] based on *Heapsort* [CLP93]) and reg (comprising all other measures [PM92]).

2.1.2 Measure-Optimal Sorting Algorithms

Given the previously mentioned measures of unsortedness, it is time to think about optimal sorting algorithms. The ultimate adaptive algorithm would obviously be reg optimal (though

there is no know algorithm like that [EW92]). A good tradeoff would be, to cover multiple measures by one algorithm. But what does it mean, to "cover" a measure; which complexity does an algorithm have, if it is optimal for a single given measure m?

Sorting algorithms have different partial problems: The problem of acquiring information and the problem of transporting data [OW12]. Sort keys have to be identified among others; the simplest way to this is to compare two keys. General sorting algorithms (so called "comparison-based sorting algorithms") are exactly based on this operation for acquiring information. Other sorting algorithms depend on input key types (see next sections) and shall not be discussed at this moment.

All sequences (all permutations) for an input length of n can be put into a binary decision tree as $n!$ leaves [OW12]. The path to a leaf of this given tree represents to path for identifying a certain sequence with pairwise comparison based operations. Given that at least all elements should be compared once for deciding about a given sequence, an algorithm needs at least n comparisons. In addition, since the decision tree is a binary tree, there are at most 2^i nodes on each level. Hence, for identifying a certain sequence in an optimal way, a logarithmic number of steps needs to be taken, which means, that a leaf should be found that is as close to the root node as possible. This means for a given sequence F, that all sequences F' of equal length that are sorted as good or better than F (according to the given measure m), will determine the optimal way for identifying F. The definition for a m-optimal sorting algorithm is, for a given constant c and a given sequence F with length n [OW12]:

$$T_A(F, m) \leq c \cdot (n + \log(|\{F' \mid m(F') \leq m(F)\}|))$$

Based on the definition of complexity classes, this formula includes a constant c that allows an additional factor. This means, a sorting algorithm is optimal to a given measure m, if it uses the least number of comparisons necessary for the given unsortedness. A small value for $m(F)$ should generate few comparisons (at least n), a high number will generate more.

2.1.2.1 Inv-Optimal Sorting Algorithms

In order to generate an *inv*-optimal algorithm, a strategy is to use sorting by iteratively inserting elements. An array cannot be used as input [OW12] here. Potentially, it is possible to find the insertion position for the next element relatively fast (for example by a *Binary Search* or an *Exponential Search*, see [OW12]), but the insertion of an element requires moving many other elements around. A theoretical suitable data structure is a "dynamic, sorted list" [OW12]. A linear, sorted linked list with an index pointing to the list's end, actually allows to insert elements

in a constant time, but it takes at maximum $O(n)$ steps in order to find an element. A data structure with inserting and finding elements in a short amount of time is an AVL tree. Here, finding also only takes $O(\log n)$ time.

Sorting linked lists is one discipline in finding measure-optimal sorting algorithms. This thesis, though, will deal with sorting data in an array structure as input. Nevertheless, it shall be given a brief overview over "close to the lower bound optimal" algorithms:
AVL-Sort [OW12] [Meh88] [Elm04] can sort *inv*-optimally in the following time bound [OW12]:

$$T(F) = O\left(n + n \cdot \log\left(1 + \frac{inv(F)}{n}\right)\right)$$

This means, that an "adaptive algorithm is optimal with respect to the number of inversions when it runs in $O\left(n \cdot \log \frac{Inv(X)}{n}\right)$" [Elm04] (see also [Gui+77]). The upper bound for the maximal *inv* number of $n(n-1)/2$ then has a complexity of $O(n \cdot \log n)$.

This way, the theoretical way of achieving an optimal strategy for the *inv* measure has been accomplished. Nevertheless, tree based solutions often lag links to practical implementations. Large overhead can be a strong barrier:
One solution [EF03] uses near optimal trees [AL90], "which is a practically complicated structure that involves a large maintenance overhead. [..] [S]plits, combines, coalescing and reducing operations [..] [make] the algorithm not fully practical." [Elm04]. "[M]ost promising from the practical point of view" [Elm04] are the adaptive algorithms *Splitsort* [LP91], *Adaptive Heapsort* [LP93] and *Trinomialsort* [Elm02]. In experiments, it was also shown that *Splaysort* is efficient in practice [MEP96].

2.1.2.2 Optimal Sorting Algorithms for Multiple Measures

It can be shown [EW92], that a simple *Merge Sort* that divides input sequences first into two halves and then into two subsequences that hold all even and all odd elements, is "adaptive with respect to Inv, Exc and Rem and optimal with respect to $Runs$, Dis and Max" [EW92]. Nevertheless, this algorithm requires a lot of overhead, which makes it hard to take the adaptive wins into account.

Listing 2.1 Pseudocode of *Odd-Even Straight Merge Sort* [EW92]

```
1   procedure OddEvenStraightMergeSort(X)
2       if not sorted(X) then
3           OddEvenStraightMergeSort( X_even_l )
4           OddEvenStraightMergeSort( X_odd_l )
5           OddEvenStraightMergeSort( X_even_r )
```

```
6            OddEvenStraightMergeSort( X_odd_r )
7            Merge( X_even_l, X_odd_l, X_even_r, X_odd_r )
8        end
9    end procedure
```

This algorithm is referred to as *GenericSort* for expected worst-case scenarios. There are other classes of measure-optimal sorting algorithms in this scenario like *Cook-Kim division*, *Partition Sort* and *Exponential Search Sort* [EW92]. For "expected-case" scenarios, ones that are less pessimistic and more realistic for average input data, there are distributed and randomized versions of the previously named algorithms [EW92]. This thesis, though, shall deal with existing and highly optimized implementations of average-case scenarios with arrays as data structure and specifically — in case of *Straight Insertion Sort* — for best-case scenarios.

2.2 Lower Worst-Case Complexity Bound for Sorting

It was introduced, what being optimal for a measure of unsortedness means, what adaptiveness is. But what is a theoretical complexity bound for worst-case scenarios? For the length of n input elements, there will be $n!$ permutations. Talking about general, comparison-based algorithms, every one of those has to distinguish between all these permutations. It is a question of *yes* and *no*. Based on information theory, the number of yes/no decisions for distinguishing between $n!$ many cases is $\log_2(n!)$ [Lan12]. Deducing an approximation, no comparison-based sorting algorithm can be faster than $\mathrm{O}(n \cdot \log n)$) in worst-case. It will be proportional to this complexity with an additional constant factor. The deduction looks as follows [Lan12]:

$$\begin{aligned} n! &\geq (n/2)^{n/2} \\ \Leftrightarrow \log(n!) &\geq \log((n/2)^{n/2}) \\ &= n/2 \cdot log(n/2) \\ &\in O(n \cdot \log n) \end{aligned}$$

2.3 Adaptive Merging

A regular merge sort uses a straight division into subsequences of equal length. Imagining that input data is almost sorted (for example to the *inv* measure, with short distances), a merge of sorted divided subsequences should not have much overlap. Nevertheless, the reguar *Merge*

Sort algorithm does not consider this heuristic. Until 1993, two attempts were made taking advantage of this little merge overlap [EW92]. A first attempt was *AdaptMerge* [CLP93]. Using this algorithm, a *Natural Merge Sort* was obtained that was *dis*-, *exc*-, *rem*-, and *runs*-optimal. "Unfortunately, AdaptMerge represented its output as a linked list of sorted segments" [EW92], which lead to trouble in implementing it performant. As a result, a second attempt involved the implementation of a *Merge Sort* algorithm that was based on straight division for the divide phase and newly defined adaptive merge. This sorting algorithm finally had little overhead was *dis*- and *runs*-optimal and adaptive for the measures *rem* and *exc* [EW92] [Van91]. The newly introduced merge algorithm worked as follows: "Let $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ be sorted sequences stored in an array with X before Y. Assume n is known and the goal is to merge X and Y. Their overlap (given by indexes l and r such that $x_l \leq y_1 < x_{l+1}$ and $y_r \leq x_n < y_{r+1}$) is first found. Second, the smaller of $x_l, \ldots, x_n$ and $y_l, \ldots, y_n$ is copied to another array and then merged with the larger sequence in the original array." [EW92].

Nowadays, for example STL's `std::stable_sort` makes use of an adaptive merge and this thesis will make explicit use of the adaptive merge implementation `std::inplace_merge`. For further details, see section 6.5.4.

2.4 Adaptive CPU Sorting Algorithms

2.4.1 Bubblesort and Cocktailsort

One of the most intuitive (and at the same time naïve) approaches is *Bubblesort*. In a very basic idea, two loops go through the input data from left to right, swapping an unsorted element right-ways to its sorted position. The complexities are [OW12]:

$$C_{max}(n) = n(n-1) = O(n^2)$$
$$M_{max}(n) = \sum_{i=1}^{n-1} 3(n-i) = O(n^2)$$
$$C_{avg}(n) = M_{avg}(n) = O(n^2)$$

The high average complexity is *Bubblesort*'s downside. If elements have long distances from their actual sort position (seen from the right side of the array), this algorithm is very slow. Each time elements have to be swapped with a maximum distance of n, starting again at the start with the next element. This means, that a few unsorted elements with a high sorted distance will lead

to a high time complexity (for more information, see chapter 6.5.1). An improved alternative is *Cocktailsort* (which is also known as *Shakersort* [OW12] [Big+08]). In each iteration, it first loops from left to right, then, from right to left. This way it becomes position-independent of unsorted elements. Both algorithms have a best case complexity of $O(n)$ and a worst case complexity of $O(n^2)$.

2.4.2 Straight Insertion Sort

Straight Insertion Sort is an adaptive algorithm. It benefits from the knowledge about presorted data ranges by insertion unsorted elements into previously sorted data. It iterates once from left to right through the data and inserts elements at the correct position to its left by swapping previous elements right. In general, in the theoretical best case the first element would have the value ∞ (or at least bigger than all other elements), so that the swapping loop would not have to check the index for the lower boundary in every iteration. In the general case, this cannot be provided, though. Since *Straight Insertion Sort* will be used within *AdaptiveFrameSort* and should provide a very good best-case behavior (it should be fast for almost pre-sorted data), the algorithm was slightly modified in order to have very few minimum memory movements (while maintaining the average case complexity). Through an analysis, the following runtime complexity's can be obtained:

$$
\begin{aligned}
C_{min}(n) &= n - 1 \\
C_{max}(n) &= 2 \cdot \sum_{i=1}^{n} i = O(n^2) \\
M_{min}(n) &= 0 \\
M_{max}(n) &= \sum_{i=1}^{n} (i + 1) = O(n^2)
\end{aligned}
$$

C refers to the number of comparisons and M to the number of data movements here. Note that the unimproved version, as it can be found in [OW12], has a M_{min} value of $2(n-1)$ (for further information, compare there). The algorithm is directly dependent on the number of inversions residing in the data, which is expressed by the *inv* measure. An average case scenario means, to expect an average — a middle — number of inversions. The runtime complexity is the following [OW12]:

$$M_{avg}(n) = \sum_{i=1}^{n} \frac{i}{2} = \Theta(n^2)$$

The previously mentioned *AVL-Sort* [OW12] is not only adaptive but optimal regarding the *inv* measure. In the concrete example of *Straight Insertion Sort* it means, that finding the insertion place for each element in the previously sorted range can be solved better, for example by a *Binary Search* with complexity $O(\log n)$ [OW12] [Lan12]. Since the elements in an array have to be swapped right anyway, the search cannot be done faster than linear. The following pseudo code snippet demonstrates the algorithm as used within *AdaptiveFrameSort*.

Listing 2.2 Pseudocode of *Straight Insertion Sort*, slightly improved standard version for having less write operations, used in *AdaptiveFrameSort*.

```
1   procedure Straight Insertion Sort(a : array, n : length)
2       for i=1 to length-1 inclusive do
3           j = i-1
4           if a[i] < a[j] then
5               temp = a[i]
6               do
7                   a[j+1] = a[j]
8                   j = j-1
9               while j >= 0 and temp < a[j]
10              end do
11          end if
12      end for
13  end procedure
```

2.4.3 Shell Sort

Based on the idea of *Straight Insertion Sort*, *Shellsort* sorts elements by insertion them at their correct position. As improvement, it tries to decrease the disadvantage that elements often have to be moved along long distances. The sequence is divided into several subsequences which will be sorted separately. Subsequences are based on a *gap distance* that will determine the indices for each element that is part of that subsequence. In every step, the *gap distance* will be decreased until it is 1. A *Shellsort* with a *gap distance* of 1 is equal to a regular *Straight Insertion Sort*. Concretely, for each iteration step h_i with $t \geq i \geq 1$ (with a chosen upper *gap distance* t) there are subsequences S_j with $1 \leq j \leq h_j$; the elements for each subsequence occur at $j, j+h_j, j+2h_j$

and so on [OW12]. In a general case, for a *gap distance* of 4 in the sequence $S_n = \{0, \ldots, n\}$ the elements for the first subsequence are 0,4,8 and so on. The second subsequence contains the elements 1,5,9 and so on. Following this approach, each *Insertion Sort* step involves little movement of unsorted elements. Nevertheless, finding an appropriate upper *gap distance*, while keeping the overall element movement small, has become a discipline on its own. The AVERAGE complexity for *Shellsort* strongly depends on the chosen increments. For example, it can be shown that the algorithm has an average complexity of $O(n \cdot \log^2 n)$, if increments have a form of $2^p 3^q$, being smaller than n [OW12] [Knu75]. Worst case complexity of this algorithm is $O(n^2)$.

2.4.4 Natural Merge Sort

As described earlier in section 2.1.2.2, *Merge Sort* works through the divide-and-conquer method. It subdivides a left and right half recursively until the length of one. After dividing each step, it merges both halves. This algorithm has a runtime complexity of $O(n \cdot \log n)$, as well for comparisons and movements in best and worst case [OW12]. Potential for becoming adaptive is the fact that the standard algorithm thinks of run lengths of exactly one element. In real-world data, run lengths are often longer, though. By searching for run lengths first, and then merging these, the algorithm will become adaptive towards the *runs* measure as well. Obviously, if the whole sequence is sorted already, the algorithm deals with one run and will not merge anything. The following complexities apply [OW12]:

$$
\begin{aligned}
C_{min}(n) &= O(n) \\
C_{avg}(n) &= C_{max}(n) = O(n \cdot \log n) \\
M_{min}(n) &= 0 \\
M_{avg}(n) &= M_{max}(n) = O(n \cdot \log n)
\end{aligned}
$$

2.4.5 Adaptive Heap Sort

Adaptive Heap Sort is based on *Heap Sort* [SS93]. It takes advantage of the introduction of a new measure of pre-sortedness, *osc* (which is mentioned earlier in section 2.1.1.4 already). The authors describe this measure on a geometrical base [LP93]. They map "each element x_i onto the point (i, x_i) in the plane and draw edges between points that correspond to consecutive elements in [the sequal] X. This gives a polygon chain corresponding to X. Intuitively, the new measure, *Osc*, tells how much the polygon chain oscillates." [LP93]. *Heap Sort* builds a heap consisting

of n elements and extracts maximum elements n times. The number of elements in each of this operation takes $O(n)$ time at most and the extraction is logarithmic in the size of the heap. Therefore, *Heap Sort* has a complexity of $O(n \cdot \log n)$ [LP93]. *Adaptive Heap Sort* uses a heap, too, but does not store all elements in the heap; instead, it only stores candidates that "can possibly be the maximum of the remaining elements" based on a *Cartesian Tree* rather than a simple *Binary Heap* [LP93]. As *Heap Sort*, *Adaptive Heap Sort* has a worst-case complexity of $O(n \cdot \log n)$ (which is complexity-optimal, see section 2.2). In a best-case scenario, though, it is faster [LP93].

2.4.6 Smoothsort

Smoothsort is a *Heap Sort* variation using a custom heap based on *Leonardo numbers* rather than the default *Binary Heap* [Dij82]. For initially nearly sorted input data, it comes close to a complexity of $O(n)$.

2.4.7 Splaysort

Splaysort is an adaptive algorithm that is based on *Splay Trees*. For sorting data, the algorithm initializes and empty *Splay Tree*, inserts each item (in input order) into the tree and through traversing the *Splay Tree* in *inorder* it receives the sorted data. As mentioned before in section 2.1.2.1, in experiments *Splaysort* was shown to be efficient in practice [MEP96]. For small, randomly sorted input data (around 16,384 items), it proved to be to constant factors slower than *Quicksort* and *Merge Sort*, but for larger input data (around 65,536 items), the constant factors decreased, based on overhead for data movement, that does not occur in the tree/pointer based *Splaysort*. For nearly sorted input data (based on the measures *inv*, *runs* and *rem*) *Splaysort* was more efficient than the compared algorithms. In an analysis [MEP96] *Quicksort* takes up to $O(bn \cdot \log n)$ time complexity, whereas the pointer based *Splaysort* takes $O(n \cdot \log n + bn)$, which is comparable to *Natural Merge Sort* with the same complexity.

2.4.8 Timsort: An Example of Hybrid Algorithms

Timsort is a hybrid sorting algorithm. It is based on *Insertion Sort* and *Natural Merge Sort* [Pet02a] and has been Python's standard sorting algorithm since version 2.3 [Fou14]. The author complained that implementations of scientific algorithms often do not care about constant factors; he wanted to implement an O-complexity adaptive algorithm that assumes real-world data rather than random data [Pet02b]. A first step is to find subsequences according to the *runs* measure.

Descending sequences will be swapped in order to be ascending. For runs with a length below 64 elements, the algorithm will use *Insertion Sort* to sort these [Pet02a]. For runs bigger 64 elements, the algorithm divides the length by a number between 32 and 64 (the minimal subsequence length), so that the resulting length of subsequences is as close to a power of 2 as possible. This way, merges are very well balanced [Pet02a]. Subsequences with a length smaller than the minimal subsequence length will be "merged" by *Insertion Sort* in order to produce the minimum subsequence length. As a follow-up, all subsequences will be merged. The merge is adaptive and very fast in a best-case scenario, since, for instance, for finding merge positions, it uses an *Exponential Search* [McI93]. The algorithm's worst-case complexity is $O(n \cdot \log n)$ and the best-case complexity is $O(n)$ [Pet02b].
Similar to *Timsort*, the algorithm presented in this thesis is a hybrid algorithm and makes use of existing algorithms in order to use their combined strength. Through the algorithms' similarities, a to C++ ported version of *Timsort* (refer to [Fuj12]) will be used for runtime comparisons throughout this thesis.

2.5 GPU Sorting Algorithms

In the last decades, several techniques for parallel programming on a GPU (so called *GPGPU*, General Purpose computing on GPUs) have arisen: OpenCL [Khr14] and NVIDIA CUDA [NVI14a], for example, offer a C/C++ based programming interface to execute code on the GPU rather than using the graphics API (such as OpenGL or DirectX API). This thesis will focus on the use of NVIDIA CUDA. "Threads are organized in a hierarchy of grids, blocks and threads, which are executed in a *SIMT* (single-instruction, multiple-thread) manner; threads are virtually mapped to an arbitrary number of streaming multiprocessors (SMs) through warps." [Ye+11]

CUDA uses several types of memory, such as register, constant memory, shared memory, local memory, and global memory. These memories have different characteristics and for achieving reasonable parallel speed-ups, fast memory has to be exploited correctly. So, for example, the previous list of memories is sorted according to speed, fastest first and slowest last. As is it typical for memory hierarchies, the fastest memory is the smallest, and the slowest memory provides most space.

Generally, parallel sorting algorithms can be divided into two categories [Gov+05a]:

Partition-based Sorting

"First, use partition keys to split the data into disjoint buckets. Second, sort each bucket independently, then concatenate sorted buckets." [Ye+11] A problem is to deal with load balancing among all the processors.

Merge-based Sorting

"First, partition the input data chunks of approximately equal size and sort these data chunks in different processors. Second, merge the data across all processors." [Ye+11] A problem is, that this type only performs well for a small number of processors.

In general, *Quicksort* usually is referred to as a fast sorting algorithm on CPU, but implementing it efficiently on GPU turns out to be rather tricky (see also chapter 2.5.5). In fact, *Mergesort* is the most widely used sorting algorithm on GPU (see also chapter 2.5.2) [Ye+10].
In the following course, the major general CUDA based algorithms will be described.

2.5.1 Bitonic Sort

Bitonic Merge Sort is a parallel sorting algorithm. Originally introduced by Batcher [Bat68], it is based on the idea of *Sorting Networks*. As a main different to comparison-based sorting algorithms, the sequence of comparisons for *Sorting Networks* is set in advance, independent on outcomes of previous comparisons. This is useful for a parallel execution. *Bitonic Sort* has a parallel complexity of $O(\log^2 n)$ passes [Pur+03].

There are multiple approaches for implementing this algorithm. Straight-forward implementations were done in [Pur+03] and [Kap+00]. Usually, sorting rates for bitonic sorters decrease the larger input arrays get (compare to [PSL10]). A good practical implementation [Ye+10], that is competitive to *Quicksort* [CT08] for large arrays (although overhead still remains) can be found in [Cap+09]. "Warpsort" [Ye+10] is another high performance implementation that profits from CUDA characteristics. For array lengths roughly up to 2^{17}, an optimized *Bitonic Sort* [PSL10] proved faster than a fast implementation of *Radix Sort* [SHG09] that can be found in the parallel Thrust library. Based on *Adaptive Bitonic Sort* [BN89] and *Bitonic Trees*, *GPU-ABiSort* could achieve a complexity of $O(\log n)$ [GZ06]. *Bitonic Trees* make *Adaptive Bitonic Sorting* inefficient for the use in hybrid algorithms; trees have to be converted into arrays and vice-versa. [PSL12] overcomes this barrier and presents a hybrid algorithm based on *Bitonic Sort* and an array-based *Adaptive Bitonic Sort*.

The next paragraph will explain the functionality of a *Bitonic Sorting Network*.
Let X be a sequence with $x_0 \leq \ldots \leq x_k \geq \ldots \geq x_{n-1}$ for some $k, 0 \leq k < n$. Then X is called a *Bitonic Sequence*. This means, that a *Bitonic Sequence* is a sequence where one part is monotonically ascending and the following part is monotonically descending (or vise-versa) (see also [Bat68]). Sequences also count as bitonic, if they can be ring-shifted into one of the previous definitions (see [PSL12]). Sorting with *Bitonic Sequences* means, that once a *Bitonic Sequence* was established, it has to be merged. This happens pair-wise for each element out of the separate subsequences. A *Bitonic Sorting Network* consists of n wires, $\log n$ phases and

altogether $\log^2 n - 1$ passes, where each pass has a sorted ascending or descending subsequence as output. Figure 2.1 clarifies such a network.

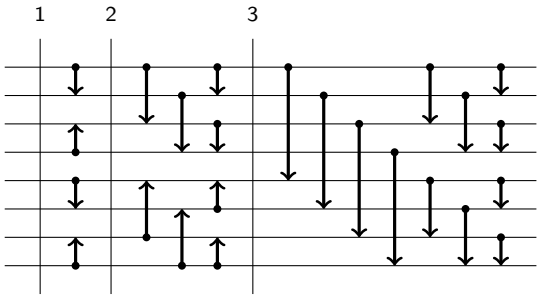

Figure 2.1 A Bitonic Sorting Network for eight elements with three phases

For a better understanding of parallel architectures, a few ideas and improvements that are CUDA specific shall be presented in the following paragraph, based on [PSL10].

A *Bitonic Sorting Network* consists of $\log n$ phases. An array of length $n = 2^k$ can be sorted by $\log n$ kernel launches with 2^{k-1} threads, each thread processing one compare/exchange operation. This way, global memory has to be accessed too often, data has to be fetched in every kernel launch. An improvement is to have threads do exchanges that exceed boundaries of their initial phases, gathering independent data that does not need results of other phases. A thread that processes these phases can keep elements in its registers and the total number of kernel launches decreases.

Bitonic Sort introduced by Batcher can only sort sequences with the length being a power of 2. Introducing a padding with *max*-values is a straight-forward solution. An input sequence of length $2^k + 1$ would result in sorting a sequence of length 2^{k+1}, though. The algorithm can be modified the way that the one existing subsequence with normal values and *max*-values will be sorted in an ascending way. This way, *max*-values will never be moved and do not have to exist physically.

2.5.2 Merge Sort

The method of splitting sequences into two halves in *Merge Sort* (see also section 2.1.2.2) offers a great opportunity for parallel algorithms. Having implemented a *Parallel Prefix Sum* ("Scan") operation [HSO07], [SHG09] demonstrates a highly efficient *Merge Sort*. For single CUDA blocks, it uses *Bitonic Sort*, for merging the algorithm splits blocks based on multiple input elements and "finding final positions of elements in the merged sequence can be done efficiently using parallel binary searches" [SHG09]. This algorithm turned out faster than the fast graphics API-based

implementation *GPUTeraSort* [Gov+05a]. Nevertheless, referring to [Ye+10], during compare-and-swap operations half of the threads remain idle and splitting operations take about $\frac{1}{5}$ of the total execution time.

2.5.3 Odd-Even Merge Sort

The idea of *Odd-Even Merge Sort* (see also section 2.1.2.2) is to first "sort all odd and all even indices separately and then merge them" [KW05] (see also [Bat68] [Sed98] [Knu75]). Overall, a complexity of $O(\log^2 n)$ holds [Bat68]. The algorithm is presented as a simple and straightforward implementation for graphics APIs on GPUs.

2.5.4 Radix Sort

Involving *Parallel Prefix Sums* [HSO07], it is possible to create a split-based *Radix Sort* algorithm [Ye+10]. A histogram-based algorithm was created in [Le 07] and [He+07], though the algorithm did not involve efficient use of memory bandwith and showed uncompetitive for large arrays, still [Ye+10]. *Segmented Scan Primitives* [Sch80] are introduced in [Sen+07]; they generalize a parallel scan (see [KW05]) by allowing arbitrary partitions (segments) [Sen+07]. Utilizing them, *Radix Sort* became faster than the split-based version in [HSO07]. The fastest *Radix Sort* presented so far can be found in the Thrust library in NVIDIA's CUDA SDK [SHG09].

The following paragraph gives a short insight on how *Radix Sort* works.
Unlike the other presented algorithms, *Radix Sort* is a non-comparative integer sorting algorithm. It sorts by consecutively comparing digits (or digit groups, the radix) and sorting them into corresponding buckets. For each digit, the runlength has a complexity of $O(n)$. The maximum element determines how many iterations it will take to have a sorted list of elements, going through each bucket from the beginning. The runtime complexity for *Radix Sort* therefore is $O(n \cdot \log w)$, where n is the number of elements and w is the number of digits [Lan12]. Note that a digit is a bitsequence with a maximum value of the chosen base, the radix. The number of digits, of course, depend on the data and the complexity's logarithm part is actually $\log_b w$, where b is the chosen base. This means, that depending on the data and the chosen base (which is also the number of buckets), *Radix Sort* can have a runtime of $O(n)$.

2.5.5 Quicksort

As in *Radix Sort*, *Segmented Scan Primitives* are introduced in a *Quicksort* algorithm by [Sen+07]. *Quicksort* has a complexity expectancy of $O(n \cdot \log n)$, but nonetheless the presented algorithm has a poor performance: On an array of four million 32-bit integers, the algorithm took 2,050.3 ms for sorting, whereas its CPU version took only 908.8 ms. The authors state that this is based on a lengthy program for managing shared memory and orchestrating the segmented scan, resulting in a large number of active registers.

[CT08] introduces a practical *Quicksort* implementation based on CUDA. It is "a practical alternative for sorting large quantities of data" [CT08] and competitive to [SA08] (see [Ye+10]). Subsequences smaller than 1,024 elements are sorted by *Bitonic Sort*.

2.5.6 Hybrid: Bucket Sort + Merge Sort

This hybrid CUDA algorithm [SA08] achieves fast sorting times through an initial parallel *Bucket Sort* splitting the list into sublists followed by a *Merge Sort*. It turned out to be more than twice as fast as *Bitonic Sort* utilizing a complexity of only $O(n \cdot \log n)$. For analysis, resulting lists of numbers of elements in each buckets are copied back to CPU memory, which leads to a small overhead in memory copying. Although the algorithm showed overall faster than *Radix Sort* in [Sen+07], it is worth to mention that it was tested on graphics cards with CUDA compute compatibility 1.1; later, this thesis will show that newer CUDA compute compatibilities (and their corresponding hardware) are much faster and, for example, memory copying times between global memories represent a real bottleneck. In addition, the algorithm uses *Bucket Sort* and therefore is only capable of sorting `float` data.

2.5.7 Others

There are different other implementations available on GPU, like the *Randomized Sample Sort* in [LOS10] that (on the the Tesla architecture) showed faster than Thrust's *Merge Sort* in [SHG09]. Nevertheless, through the random selection it had bad load balancing [Ye+11]. Other parallel algorithms involve *External Sorting*, database sorting and parallel architectures other than GPUs, which shall be not part of this work.

2.6 Adaptive GPU Sorting Algorithms

2.6.1 Odd-Even Sort

Based on *Bubblesort* and *Cocktailsort* (see section 2.4.1), *Odd-Even Sort* [Dom11] [KW05] is a simple implementation for *Sorting Networks*. It works by first sorting all odd indices, then all even indices. This will iterate at maximum $\frac{n}{2}$ times until the full array is sorted. The advantage is, that it can be very easily parallelized: All odd indices can be sorted by a single CUDA kernel launch, as well as all even indices. Figure 2.2 presents the algorithm as a *Sorting Network*:

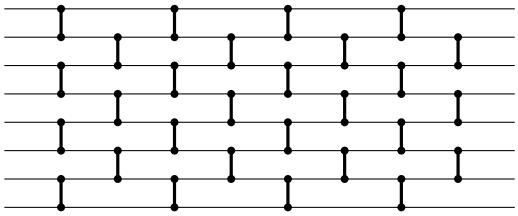

Figure 2.2 An Odd-Even Sorting Network for eight elements with eight phases

A downside of this algorithm is the huge overhead for at maximum n kernel launches and the worst-case complexity of $O(n^2)$. A positive side is the adaptiveness and best-case complexity of $O(n)$. Later in this thesis, all three GPU algorithms will be very closely analyzed for the best-case scenario in paragraph 6.5.2.

Chapter 3

Methodology

In animations with a rigid and a deformable object, it seems likely that sometimes parts of the scene do not move, if you look at them from one side. For example, objects might be moving only away from your perspective, which means, they do not appear to be moving, rather changing their size. As a thought, this occurrence should be visible in *Axis-Aligned Bounding Boxes*. Keeping this practical example in mind, the overall task for the thesis was to create a sorting algorithm, that takes advantage of pre-sorted data and is capable of running on GPU. As a follow-up, taking an existing implementation of ABiSort [GZ06] raised the question of which kind of measures of unsortedness the algorithm could take advantage of (the theoretical background was described in chapter 2.1 already). The idea was to analyze real world data according to the levels of unsortedness that they provide and analyze the algorithm to find matching values. As simplification, in the previously mentioned collision detection, the AABBs were reduced to one axis only, specifically, triangles' bounding boxes were projected onto the X-axis. A good analysis of unsortedness in this manner will be presented in chapter 5.1.

As it turned out, in a *Bitonic Sorter*, pure values of unsortedness do not allow to conclude any possibility for adaptiveness. Specifically, *runs* have to be identified at their corresponding position, in order to cut off whole merge branches or to take advantage of pre-sortedness in already existing bitonic sequences (in case one node of a merge branch is sorted in a descending way already, its other node only has to be sorted in an ascending way). This leads to the idea of implementing an interface that can visualize the examined scenes according to their levels of pre-sortedness. This will be explained in section 5.2. There, bounding boxes are put into relation to their corresponding global position within the scene.

The preceding methods of analysis lead to two different methodologies for creating an adaptive algorithm: Visual analysis of pre-sortedness involves *Trial and Error* methods to find ways how to exploit them. In a very early state, this method made clear that rather than adjusting a time-local algorithm, it would be easier to implement a new algorithm that could make use of bounding boxes that would change only slightly from frame to frame; in fact, this seemed to be the case in many of the analyzed scenes. This idea became the main objective for this work ever

since. Pre-sorted data ranges were selected based on different parameters which were adjusted based on repeating *Trial and Error*, sometimes visually and sometimes via *Brute Force*, sorting scenes with a variety of possible parameter combinations and analyzing the adaptiveness (for example, based on speed analysis in ratio to the corresponding measures of unsortedness).

Later, these parameters were selected in a more generalizing way, scene-independent: Measures of unsortedness turned into the main driving forces for setting parameters. The in this thesis presented novel algorithm's overall evolution can be found in chapter 6.

Chapter 4

Scenes

4.1 Selection of the Scenes

Selecting the test scenarios, it was important to bear in mind that scenes should be selected that are also used in other papers, in order to have uninfluenced scenarios, ones that are not set up to fit for the algorithm. The UNC Dynamic Scene Benchmarks collection [UNC14] is a source that offers multiple pre-animated frame sequences that are used in [MZ14], for example.

A general prerequisite for sorting bounding boxes adaptively is that the data contains some areas with little or no changes. Practically this means, that an animation scene should have either a high time resolution (meaning a big frame rate, which is incorporated in the funnel scene and can be found in section 4.2.2) and/or a high amount of triangles (where chances might be bigger that there are some areas with little scene changes). In general, a combination of deformable objects and rigid objects can lead to adaptive collision detection based on bounding boxes. Based on these assumptions, a counter example from the UNC collection is the "dragon and bunny" model; despite its high amount of triangles (~252,000) it is hard to use it in the boundaries of this thesis, since it only has 16 frames and changes are to quick to imply pre-sorted data.

4.2 Low Polygon Scenes

The following two scenes can be obtained from [UNC14].

4.2.1 Clothball

The clothball scene has 46,598 vertices and 92,230 triangles (which results in 184,460 bounding box elements to be sorted). Altogether there are 94 Frames in which a deformable cloth falls

down onto a rigid ball. Afterwards, it turns an wraps itself around the ball.

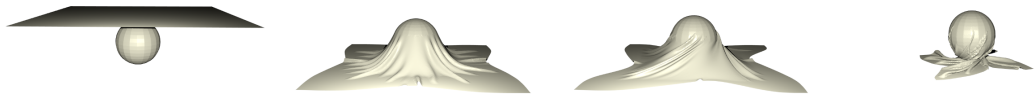

Figure 4.1 Clothball scene

4.2.2 Funnel

The funnel scene has 9,450 vertices and only 18,484 triangles, which yields a list of 36,968 bounding box elements. The low amount of triangles is opposed to a high number of frames: 501. This high time resolution offers a chance for adaptive sorting, especially noting that the scene contains two rigid objects (a funnel and a plane) and two moving and deforming objects (a ball that drags a deformable cloth down through the funnel).

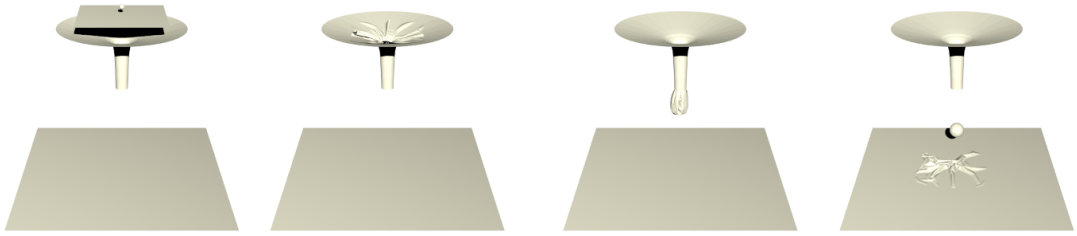

Figure 4.2 Funnel scene

4.3 High Polygon Scenes

4.3.1 Clothcar

As high polygon scene the clothcar has 365,362 vertices and 692,156 triangles. Altogether there will be 1,384,312 bounding box elements. Influenced by gravity and a strong wind, in 151 frames a deformable cloth with 172,992 triangles falls onto a car with 519,164 triangles. The car model can be obtained from [Ble14], the cloth and its animation were added afterwards by me.

Figure 4.3 Clothcar scene

Chapter 5

Analysis

5.1 Scenes' Analyses

In the following course all three scenes will be analyzed using the three measures of pre-sortedness according to [OW12]: *inv*, *runs* and *rem*.
It is notable that for the two fully analyzed scenes the *inv* measure will never grow much on a scale expressed as percentage (in this case, not beyond 2.14%). As for the other two measures, the visual blend of triangles (in X-direction) can roughly be deduced as well. As long as the cloth does not touch the ball in the clothball scene (which are the first four frames), all of the three measures of unsortedness are relatively low (below 3%), including the measures *runs* and *rem*. Towards the end, the more the cloth wraps itself around the ball, the shorter non-coherent ascending subsequences get, the *rem* measure increases.

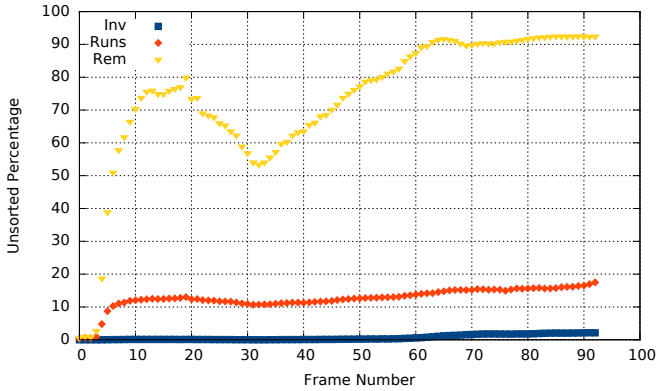

Figure 5.1 Clothball unsortedness

For the funnel scene, both measures clearly imply the frames where the cloth and the ball are located inside the funnel tube (frames 134-403).

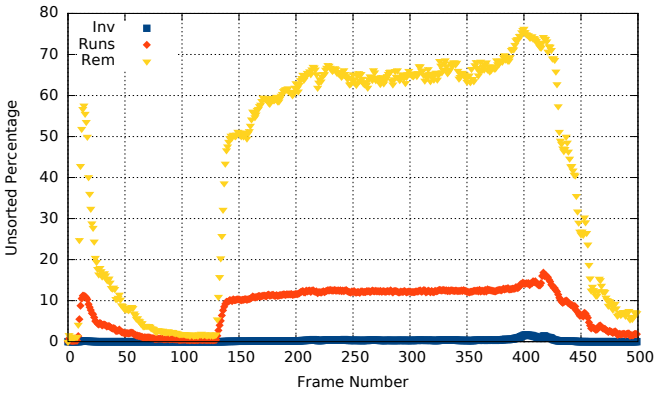

Figure 5.2 Funnel unsortedness

Based on its definition, the *inv* measure has a maximum value of $\frac{n(n-1)}{2}$ (see chapter 2.1.1.1), which results in a run complexity of $O(n^2)$ for determining this measure for a dataset. For this reason, only the first 6 frames were considered exemplary for this measure in the clothcar scene. Starting from frame 29, a moment where half of the cloth has dropped onto the car's surface, the *rem* measure stays stable at around 25%. At the same time, the *runs* measure does not exceed 5% of unsortedness. These values can imply a good possibility (compared to the clothball scene with a maximum *rem* value of 92.5% and a maximum *runs* value of 17.5% and the funnel scene with 76.3% for *rem* and 16.8% for *runs*) to sort pre-sorted subsequences adaptively.

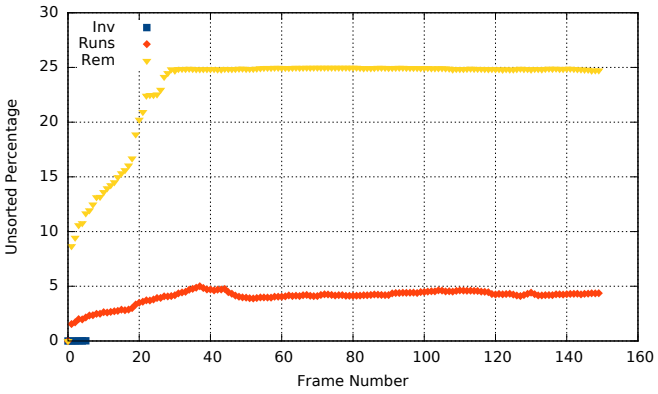

Figure 5.3 Clothcar unsortedness

5.2 The Analysis Program

For a better understanding of unsortedness levels, a program with visual output was written based on OpenGL techniques.
The scenes are being read frame by frame as ".obj" files. Later, the arrow keys help navigate through the rendered scenes.
The insertion of each frame works the following way:

- Take frame 0
- Generate bounding boxes around its triangles in X-direction
- Insert bounding boxes into the bounding box array with indices for the
  - Left side: $2 \cdot triangleIndex$
  - Right side: $2 \cdot triangleIndex + 1$
- Take frame n | n > 0
- For each bounding box entry select the corresponding triangle within the frame
- Generate bounding boxes around the triangle in X-direction
- Update bounding boxes in the bounding box array

The colors used for displaying objects are false colors: A dark blue color means no unsortedness; after being sorted in the previous frame these bounding boxes did not move; a red color means full unsortedness; after being sorted in the previous frame these bounding boxes had the maximum distance frame wide. In technical terms, the color space is HSV where saturation and value are set to 1 and the hue value will be between 240° (blue) and 0° (red):

$$\begin{aligned} saturation &= 1.0 \\ value &= 1.0 \\ hue &= 240 - \frac{distance}{maxDistance} \cdot 240 \end{aligned}$$

For the use in OpenGL the resulting HSV value will be converted into the RGB color space.
The distance is a value computed after the following scheme:

- Mark the extra distance value in each bounding box entry with its current position in the bounding box array
- Sort the bounding boxes according to their sides' positions
- The sorted distance is the sorted position minus the old distance value

The final bounding box data structure looks like the following:

int triangleIdx
: The triangle index that this bounding box item is referring to. A negative index implies a

left edge, a positive index a right edge.

float bdPos

Position of this bounding box item on the X-axis

int sortPositionAndAfterSortDistance

AdaptiveFrameSort-specific item. Used during sorting to calculate items' sort distances.

Figure 5.4 shows an early state of the clothball scene. The colors chosen are false colors referring to the bounding boxes' sort distances, and in pink, triangles of the previous frame are visualized, that have a distance of at least 80% of the maximum distance in this frame.

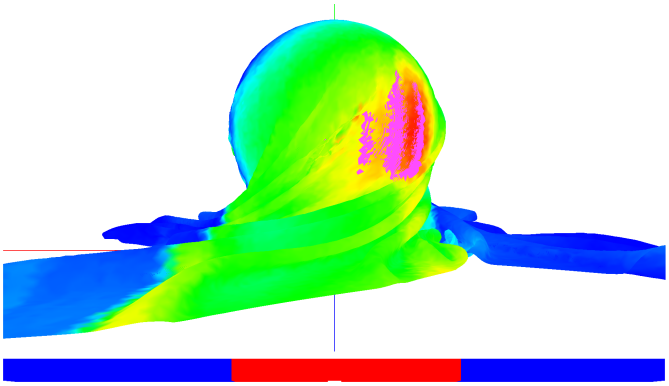

Figure 5.4 Early state of the clothball scene. False colors represent frame-wide unsortedness of bounding boxes in the model (compared to the previous frame); red triangles represent high unsortedness, blue triangles low unsortedness. The bar undearneath the model visualizes calculated sort ranges: blue ranges represent n-nsquare algorithm parts and red ranges represent nlogn algorithm parts.

Underneath the object, the figure shows hypothetical *sort ranges*. The cloth itself appears to have higher sort distances in the center of the frame, around the ball that it wraps itself around. The outer edges of the cloth seem to have less movement. According to this idea, the newly developing algorithm will try to sort ranges with a relatively small maximum sort distance with an already existing algorithm that will take advantage of the pre-sortedness, in this case *Bubblesort* (the blue ranges). *Bubblesort* has a best case complexity of $O(n)$ and a worst case complexity of $O(n^2)$ (see also chapter 2.4.1). Since there are many algorithm variations, the class of these adaptive sorting algorithms shall be referred to as **n-nsquare** algorithms from now on. The other ranges will be sorted by an algorithm with a smaller complexity for the worst case, STL's `std::sort` (the red range). `std::sort` has an average and best case complexity of $O(n \cdot \log n)$ [SGI14b]. This class of non-adaptive sorting algorithms shall be referred to as **nlogn** algorithms from now on. For a frame-based analysis of sort distances refer to figure 5.5, which shows very low maximum sort distances for frame 4 (181 / 184,459 elements).

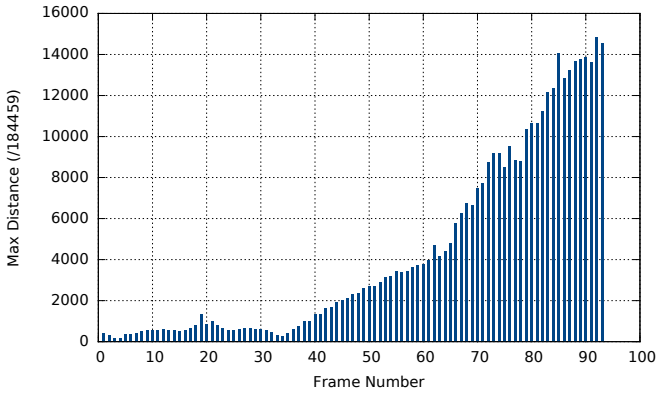

Figure 5.5 Clothball's maximum sort distances. At this algorithm state, small sort ranges might imply little unsortedness and a good speed-up chance for an adaptive algorithm.

Chapter 6

Algorithm

In this chapter, the algorithm *AdaptiveFrameSort* will be explained in detail. It is an outline to the entire evolution of the algorithm. Paragraphs contain analyses or improvements: Sections that are tagged with the name "Improvement" present final results that are incorporated into the algorithm based on previous analyses.

Chapter 6.1 presents an early state and shallow analyses, it can be referred to as a brainstorming introduction for getting grip onto the problem of adaptiveness.
In the further progress, a first adaptive GPU algorithm will be presented (chapter 6.2).
AdaptiveFrameSort utilizes existing sorting algorithms on an abstract level in order to sort data. Path-breaking improvements could be established once these underlying algorithms were analyzed thoroughly and algorithm-selection parameters were based on scene-independent, sub-algorithm specific measures. These analyses can be found in chapter 6.5.
Additional improvements of vital importance can be found in the chapters 6.6, 6.7 and 6.8.
A pseudocode will be presented (chapter 6.9.2) and the algorithm's functionality will be explained by an example (chapter 6.9.1).

6.1 A First Approach: Parameter Pair MaxDistanceThreshold and MinimumRangeLength

The in chapter 5.2 described approach of using sort ranges based on the *max* measure (see also chapter 2.1.1.4), yields two different parameters:

maxDistanceThreshold
: Maximum sort distance within a range. No data element in this range will have a bigger sort distance than this value.

miminumRangeLength

A minimum length of each sort range; otherwise ranges might be too short and the overhead for merging so many fragments is too big.

In case of the clothball scene a good distribution would be to have adaptively sortable ranges at the edges and a non-adaptive range in the middle. Based on this idea, the following parameters proved themselves as visually appropriate:

$$maxDistanceThreshold = 1,845$$
$$mimimumRangeLength = 6,400$$

A problem at this state is that these parameters are customized for the clothball scene. In fact, for the funnel scene a `maxDistanceThreshold` of 93 turned out to be a better value. As reaction, universal parameters have to be found.

The approach so far looks as the following:

1. For frame 0, generate bounding boxes on the X-axis.
2. Set the distance attribute for each array item to its current position.
3. Sort the whole frame by `std::sort`.
4. Now, the sort distance is the sorted item index minus its unsorted item index (the distance attribute's value).
5. Determine the "Sort Ranges" according to the parameters `maxDistanceThreshold` and `mimimumRangeLength`.
6. These sort ranges are a suggestion for the next frame: Sort the "relatively pre-sorted" ranges by *Bubblesort*.
   - As "safety net" include a fallback that stops the execution for *Bubblesort* in case too long distances will be sorted in this new frame.
   - In case of a fallback, use `std::sort` for the current sort range.
7. Sort all other sort ranges by `std::sort`.
8. **Cross-range items** need to be sorted. These are items that are sorted within their range but their global position resides within a different range. In order to sort them, that means to merge them into their corresponding sort ranges, execute a global *Bubblesort* across all sort ranges. This will be relatively effective, since each range is sorted already and in case of little movement there should not be too many cross-range items.

If only one sort range exists, the merge step can obviously be omitted. At this point, a performance improvement might be to introduce a keyframe based analysis of sort distances (for example every 15 frames). As it turned out, the bounding boxes in the investigated scenes move too much for this kind of idea; the algorithm would not be exact enough anymore.

6.1.1 First Adaptive Success

In a first implementation of this algorithm some functionality was implemented differently: Instead of `std::sort`, a simple implementation of *Merge Sort* was used.
In this configuration already, the adaptive algorithm was faster than *Merge Sort* for frame 4 of the cloth scene; *Merge Sort* had 27 ms of computation time, the adaptive algorithm had 25 ms of computation time (in a single run). The overall sort time for three found sort ranges took 21 ms, the global sort of the sorted ranges took 3 ms and the computation of the next frame's sort ranges took only 1 ms. This confirmed the basic concept of the algorithm.
In case of a fallback, as it happens in frame 2, the adaptive algorithm had a computation time of 48 ms.

6.1.2 A Closer Parameter Analysis

A quantitative analysis of different values for the `maxDistanceThreshold` in frame 3 of the clothball scene with a `mimimumRangeLength` of 10,000 leads to the following top times (note that a single run of *Merge Sort* takes 27 ms):

maxDistanceThreshold	Frame Sort Time (ms)
9	28
14	17
15	15

Table 6.1 maxDistanceThreshold analysis for frame 3 of the clothball scene. Too big thresholds might result in unrealistic forecasts of n-nsquare sort ranges and in high execution times; too small thresholds might not fully deploy the advantage of n-nsquare algorithms.

Taking up value 15 as best choice for the `maxDistanceThreshold`, another analysis involving different values for `mimimumRangeLength` results in the following top timings:

minimumRangeLength	Frame Sort Time (ms)
3,000	16
6,000	18
7,000	16
8,000	16
9,000	18
10,000	18
11,000	19
12,000	21

Table 6.2 minimumRangeLength analysis for frame 3 of the clothball scene. Too small range length values may lead to more n-nsquare sort ranges and overall higher execution times; too long range lengths might result in too few n-nsquare sort ranges and finally in no adaptive gain.

In order to have least merge overhead in the worst case of $\frac{arrayLength}{minimumRangeLength}$ sort ranges, the biggest of the fastest ranges will be chosen as new value for `mimimumRangeLength`: 8,000.
In summary, the closer analysis lead to a faster algorithm that improved its sort time from 25 ms (in section 6.1.1) to 15 ms.

6.1.3 Improvement: Stable Sort

One thought on how to improve the above algorithm is how to decrease the sort distances maintaining the same results. The response is simple: The usage of stable sorting algorithms will still result in a sorted array but with possibly less distances for each item. *Bubblesort* is already stable. The instable `std::sort` will be replaced by STL's stable sorting version, `std::stable_sort` [SGI14c]. This improvement will be used in the algorithm henceforth.

6.1.4 Improvement: Local Merge Heuristic

A general problem of *Bubblesort* is that it sorts more slowly the further at the array's end an item is. This was the main motivation for changing the size of the merge ranges. An additional thought though, and this is even more important for later versions of this algorithm, is that

the less movement from frame to frame happens, the fewer cross-range items will exist. This means, that the merge of all sort ranges should be adaptive itself. On CPU, this can be actually achieved (as mentioned in next paragraph). On GPU, though, the Thrust library does not offer an adaptive merge algorithm. In order to keep merge times low then also, the following approach can help:

After the sort of each frame, the maximum sort distance will be determined. Exactly this value can be used for making assumptions about the merge ranges. Taking the clothcar scenario as example, the following might be the case: Assuming there was no gravity and the wind had a constant strength, the cloth would float from the middle of the car towards the left side (on the X-axis, which is actually the front of the car model) in a constant pace. Therefore, the change from frame to frame is constant.

In case we sort random subsets of data within the frame and afterwards want to merge them into one single data set, the case is that if we put an interval around the borders of the subsets, this interval does not have to be longer than the maximum sort distance of each frame (which in this scenario will be constant).

Now in the clothcar scenario there is the influence of gravity and the cloth does not float in a constant pace. Nevertheless, we might assume that the change in its speed in x-direction will not change more than to maximal twice its speed of the previous frame. In more general animations of course the sort distances is not equal to objects' speeds. But, nevertheless, it is worth a try to introduce this assumption as heuristic into the algorithm.

So, after sorting all sort ranges, intervals will be constructed around the sort range borders with a length of $2 \cdot maxDistance$ of the previous frame into each direction (since we do not know in which direction elements might be sorted). After merging the data within these intervals, the intervals will be turned into new sort ranges. Then, a global check will be performed to tell whether the merge heuristic was successful and merged all unsorted cross-range items or not. If this is not the case, all sort ranges (including the new interval-ranges) will be merged by their full length subsequently into one. Figure 6.3 in the next paragraph demonstrates the saved times based on this heuristic. It is important to note that in case the heuristic does not succeed, the approach created an overhead for the algorithm. This is why in the current implementation, the heuristic will only apply if there are two or three sort ranges.

6.1.5 Improvement: Adaptive In-Place Merge Instead of a Global Bubblesort

In fast moving scenes, the global array sort by *Bubblesort* for "merging" cross-range items can be very ineffective, as can be seen in table 6.3. The STL library offers a fast alternative: `std::inplace_merge`. It is an adaptive merge algorithm and grows linearly to the number of elements

that need to be merged (in case there is enough auxiliary memory available) [SGI14a]. This algorithm will be used henceforth (by the CPU version) for merging all sort ranges subsequently into one.
Keeping the GPU version in mind, the Thrust library only offers a merge with an auxiliary array. For this case a special merge function was set up to merge the first and second sort range into an auxiliary array that is a copy of the current data array. The third sort range will be merged with this result into the original data array then and so forth. This swap of auxliary array and original data array leads to least additional memory use. Table 6.3 will show that including `std::merge` in this algorithm will lead to a faster merge compared to using *Bubblesort* as merging algorithm. It will also show the final state with the use of `std::inplace_merge`. Note that the values represent timings without decimal places; therefore, "zero" values imply timings between 0 ms and 1 ms.

Algorithm	Merge Time (ms)
Bubblesort	25
Bubblesort (Local Merge Heuristic)	0
`std::merge`	4
`std::merge` (Local Merge Heuristic)	0
`std::inplace_merge`	0
`std::inplace_merge` (Local Merge Heuristic)	0

Table 6.3 Merge algorithm analysis for frame 6 of the clothball scene. Number of sort ranges: 3

6.2 An Adaptive GPU Algorithm

Considering that *AdaptiveFrameSort* makes use of existing algorithms, it is directly possible now to exchange the CPU sub-algorithms by GPU versions.
In a first setup, *Bubblesort* was exchanged by the GPU implementation of *Odd-Even Sort*, `std::stable_sort` by Thrust's `thrust::stable_sort` and `std::inplace_merge` by `thrust::merge`.

These replacements do not make *AdaptiveFrameSort* fully GPU based. The part of computing the new sort ranges for the next frame still happens on CPU. This leaves the problem that input data, that resides in the VRAM, first has to be copied to the RAM in order to execute the sort range computation. For example, to copy the clothcar scene's bounding boxes with 1,384,312

elements from VRAM to RAM on the test system with CUDA compute compatibility 1.3, it takes around 6 ms. Later in this work the fully GPU based algorithm description can be found.

6.3 Improvement: Parameter Pair InvThreshold and MinimumRangeLength

Finding sort ranges according to maximum sort distances is a fast way of determining possible ranges that can be sorted by an n-nsquare algorithm. The downside though, is that the measure of maximum sort distances is no effective measure for the deployed sub-algorithms. The selected algorithm *Straight Insertion Sort* is directly dependant on the number of swaps that need to be done for sorting the input data. This is the *inv* measure. For *Cocktailsort*, this measure also applies, although it has also a certain dependency on the location of unsorted items within the input data. This extra measure is hard to calculate though and not of use for over the time changing data. So, as heuristic, it is okay for this algorithm to neglect the determination of the location of unsorted items. In addition, the use of the *max* measure can result in "false positives" for detecting n-nsquare sort ranges. In case a range consists of small sort distances but many of them, sorting it can result in many swaps (and therefore high execution times) although the maximum sort distance is low.

Therefore, a new way and — at least for *Straight Insertion Sort* to 100% according to its actual runtime and sort efficiency — a realistic evaluation of sort ranges is to determine them based on the SUM of sort distances. This sum is equivalent to the number of necessary swaps.

At the same time, a fallback implementation is similarly simple, since instead of comparing each item's sort distance with a maximum, the algorithm simply has to add the sort distances of each item. An initial threshold of 60,375 for *Straight Insertion Sort* on CPU and 250 for *Cocktailsort* on GPU is being implemented. At least for the CPU version, sort ranges from now on will most probably have a wider range as compared to the `maxDistanceThreshold`. This in turn leads to smaller nlogn ranges and therefore overall smaller sorting times. In the clothball scene, overall lower adaptive execution times could be witnessed, since sort ranges were forecasted more realistically (with less "false positives"). This is a major improvement and will be used in the final version of *AdaptiveFrameSort*.

6.4 Improvement: Individual MinimumRangeLength

So far, the minimum sort range length was restricted to 8,000 elements. Obviously enough, this length should vary for different scene lengths. Especially for bigger scenes, this value might be too small and would generate too much overhead for merging all sort ranges. At this point, a dynamic determination of minimum range lengths will be implemented. The idea is to divide each scene into a constant amount of ranges, even before the determination of sort ranges (for more understanding refer to figure 6.8). As for the final implementation, this constant number of ranges will be 20. Before this improvement, the input data was scanned through item by item, until the minimum sort range length was reached. A previous determination of ranges did not happen. In the new way of thinking, a subdivision into ranges with a fixed length beforehand, is very easy and fast to deal with also by parallel GPU algorithms. So far, the algorithm could not be ported fully to the GPU, since sort ranges were calculated item-by-item (see chapter 6.2). The Thrust library, for example, offers the `thrust::inclusive_scan` as prefix-sum function. The usage will be discussed thoroughly later in chapter 6.6. A major outcome of the introduction of individual minimum range lengths therefore is not directly a speed-up of individual scenes, but a generalization for all scenes and an algorithm that can be scaled up for the usage in parallel GPU architectures.

6.5 Sub-Algorithms' Analysis

AdaptiveFrameSort utilizes different existing sorting algorithms in order to sort data. In the following sections, these underlying algorithms shall be examined thoroughly in order to understand, how to use them optimally according to their strengths.

6.5.1 Improvement: Straight Insertion Sort Instead of Bubblesort on CPU

As table 6.3 showed, a global sort based on *Bubblesort* takes a lot of more time than its local merge heuristic version, which means the sort of a shorter array. Figure 6.1 clarifies why higher sort distances can result in higher execution times for *Bubblesort*.

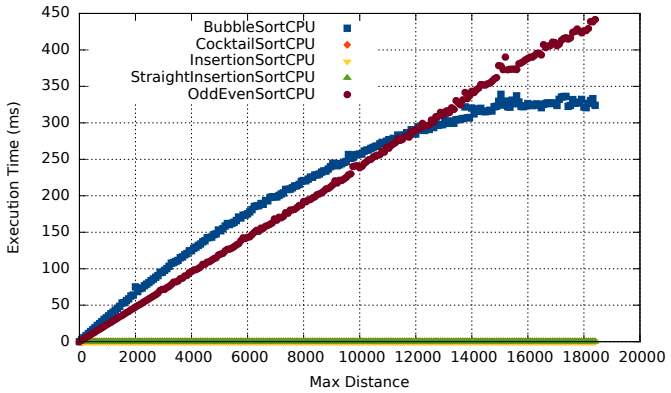

Figure 6.1 CPU in-depth analysis on an array with 18,446 elements. A pollution (an unsorted item) is inserted with an increasing sort distance towards the end of the array

As stated before in chapter 2.4.1, a downside of *Bubblesort* is sorting elements that are located at the right side of the input data, but have their sorted location at the left side of the data. As soon as the algorithm does not swap any element in one pass, it halts. This explains why its runtime is not linear: Each pass swaps the "pollution element" (which first is located at the right end of the input data) one step more to the left, reading the input data from left to right in each pass. After that pass, the algorithm halts. The closer the element gets to the left side (the start of the scan), the sooner the element will be sorted and the sooner the algorithm halts. One idea of avoiding this downside of *Bubblesort*, is to scan the input data once from left to right, and in the next step from right to left. This "bi-directional" version is called *Cocktailsort*. A yet faster implementation on CPU though is *Straight Insertion Sort*, which scans for any unsorted item and directly inserts it at its correct position. Apart from naïvely searching the array, additional overhead for scanning does not appear.
Odd-Even Sort in turn is not affected by the sort direction of elements, but has to undergo more passes the higher sort distances are. This algorithm is not relevant for the CPU, but is mentioned in this paragraph for comparison with the GPU version.

This means, that *Straight Insertion Sort* will be used as adaptive algorithm on CPU from now on. For a **pollution distance** of 18,445 (which in this example means an unsorted item that has its sorted position 18,445 elements leftward), its timing result is 0.06 ms, while *Insertion Sort* takes 0.2 ms and *Cocktailsort* takes 0.15 ms. *Bubblesort* takes 324.05 ms.

6.5.2 Improvement: Cocktailsort Instead of OddEvenSort on GPU

Similar to the CPU version, also on GPU *Bubblesort* takes more time for unsortedness at the input array's end. Figure 6.2 clarifies this behavior. Both *Bubblesort* and *Cocktailsort* were

executed with 512 threads and one block, whereas *Odd-Even Sort* ran with 512 threads and $\frac{n}{numThreads} + 1 = 37$ blocks. Only 512 threads instead of 1,024 (as typical for the Kepler architecture) were used for direct comparisons to the Tesla architecture in other tests (where the maximum of threads was 512). Not using the maximum configuration though does not affect the outcome of this test, since tendencies remain the same.

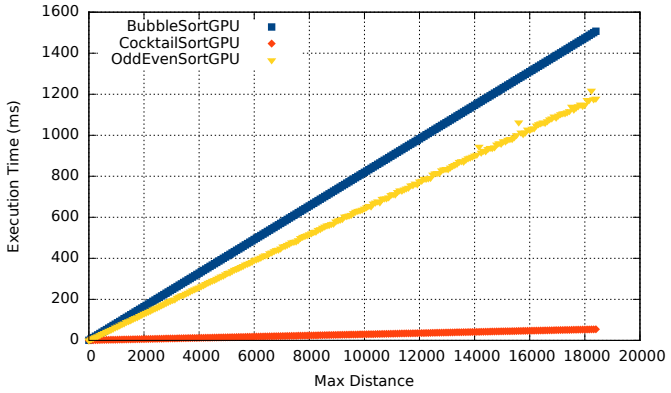

Figure 6.2 GPU in-depth analysis with CUDA compute compatibility 3.5 on an array with 18,446 elements. A pollution (an unsorted item) is inserted with an increasing sort distance towards the end of the array

This means, that *Cocktailsort* will be used as adaptive algorithm on GPU from now on. For a pollution distance of 18,445, its timing result is 54 ms, while *Bubblesort* takes 1,510 ms and *Odd-Even Sort* takes 1,179 ms.

Cocktailsort is a self-written CUDA implementation that uses one CUDA block with a maximum number of threads on it (512 on Tesla architectures, 1,024 since Fermi architectures [NVI14b]). In order to overcome overhead, it starts performing for input sizes bigger than the number of threads that it runs with. Each thread will prefetch one item from global memory into shared memory. Afterwards, each thread will compare and swap items first from left to right, and in the next step, from right to left. In case a certain number of swaps is reached (InvThreshold), the execution will stop. Whereas the execution in only one block is a general downside, it comes in handy for the use in *AdaptiveFrameSort*. Each n-nsquare sort range can be sorted fully parallel, each range belonging to one block in one kernel launch in total. This way, overhead for separate kernel launches can be neglected. In general, it is to mention that a performant implementation of this *inv*-adaptive O(n) best-case algorithm is highly crucial to the overall performance of *AdaptiveFrameSort*. Since only a small number of elements is loaded into the fast shared memory (only one thread block is available), the global memory bandwidth is a devastating bottleneck for the used implementation of *Cocktailsort*. The realization of a multi-block version of this algorithm remains difficult and unsolved within the boundaries of this thesis.

6.5.3 The Final Sub-Algorithms

Based on the previous findings, the eventually chosen sub-algorithms can be found in table 6.4.

Algorithm type	CPU	GPU
n-nsquare sort[1]	*Straight Insertion Sort*	*Cocktailsort*
nlogn sort[2]	`std::stable_sort`	`thrust::stable_sort`
Merge	`std::inplace_merge`	`thrust::merge`

Table 6.4 Final sub-algorithms. As used in this work, [1]n-nsquare refers to the class of $O(n^2)$ average case algorithms with a best case complexity of $O(n)$, [2]nlogn refers to the class of $O(n \cdot \log n)$ average case algorithms.

6.5.4 In-Depth Algorithm Comparison on CPU

The main idea behind this algorithm is to take advantage of fast sorting algorithms for pre-sorted data. In theory, the complexity for the used *Straight Insertion Sort* is O(1) (movements) and $O(n)$ (comparisons) in best case and $O(n^2)$ in worst case (see "Einfügesort"[OW12]). `std::stable_sort` uses the *Merge Sort* algorithm [Knu75] and its complexity is $O(n \cdot \log n)$ in case of sufficient memory and $O(n \cdot \log^2 n)$ in case of insufficient [SGI14c]. So, w.l.o.g., a general AVERAGE complexity is $O(n \cdot \log n)$. Note that STL's algorithm is "an adaptive algorithm" [SGI14c]. In addition, it is a hybrid algorithm (see also section 2.4.8). As of version 3.3 [SGI14d], in case of insufficient memory, `std::stable_sort` will use a regular implementation of *Merge Sort* with the use of *Insertion Sort* for subsequences smaller than 14 items (which is slightly adaptive for different input data, applying different memory copy strategies) in combination with a merge without buffer. In this work field's more common case of a sufficient amount of memory, the algorithm will use *Insertion Sort* in combination with a merge with buffer memory for smaller sequences (based on the algorithm's complexity, "small" will not be defined further here) and an adaptive merge for at least two halves inside the input array. Note that the implementation is quite complex and optimized and the overall adaptiveness depends on data conditions and memory situations. Still, the algorithm's best case complexity is slower than *Straight Insertion Sort*'s $O(n)$ complexity. In order to investigate real world conditions, a speed comparison for different scenarios with each 1,000 iterations can be found in table 6.5.

Algorithm	Sortedness	Time (ms)
`std::stable_sort`	Unsorted (inversely sorted)	12.0
`std::stable_sort`	Unsorted (random)	18.0
`std::stable_sort`	Sorted	8.9
Straight Insertion Sort	Sorted	0.3

Table 6.5 Speed comparison for adaptive data. Input array: 300,000 elements of type int

For that reason, the nlogn algorithm times in figure 6.3 refer to tests with *sorted* data which represents the worst case scenario for speed comparisons.

The respective complexity scenario for *Straight Insertion Sort* depends on the input data's level of pre-sortedness. In fact, the runtime is linearly equivalent to the number of pairwise swaps that have to be done in order to sort the input data (which can be expressed by the *inv* measurement). The runtime for `std::stable_sort` has a (in average) logarithmic dependency on the input data's length. The different dependencies unveil a vital functionality for *AdaptiveFrameSort*. In fact, *Straight Insertion Sort* will have a similar runtime for any input data's length, as long as the number of necessary swaps stays constant; merely a comparably small summand will be added to the execution time for reading the data in a linear complexity. So, w.l.o.g., let us assume that *Straight Insertion Sort* does not change its runtime for different sizes of input data's length but only according to the level of its unsortedness. In order to gain advantage from using an n-nsquare algorithm like *Straight Insertion Sort* for fairly pre-sorted data, it is important to understand where timings are faster there in comparison to an nlogn algorithm like `std::stable_sort`. For comparing real world data timings for both algorithms were collected.

Figure 6.3 shows timings for both *Straight Insertion Sort* and `std::stable_sort`, sorting data that was modified in a way so that sorting it would take the stated number of swaps. The timings for `std::stable_sort` are average values calculated from 20 iterations. The timings for *Straight Insertion Sort* refer to different number of swaps in an array of fixed length, with one iteration each (n=18,446. In order to have a rather representative array size, close to real world data, the length of two ranges in the *clothball* scene was chosen, assuming 20 is the number of ranges). This length is fixed in order to compare different number of swaps (for *Straight Insertion Sort*) and different input lengths (for `std::stable_sort`) in one graph.

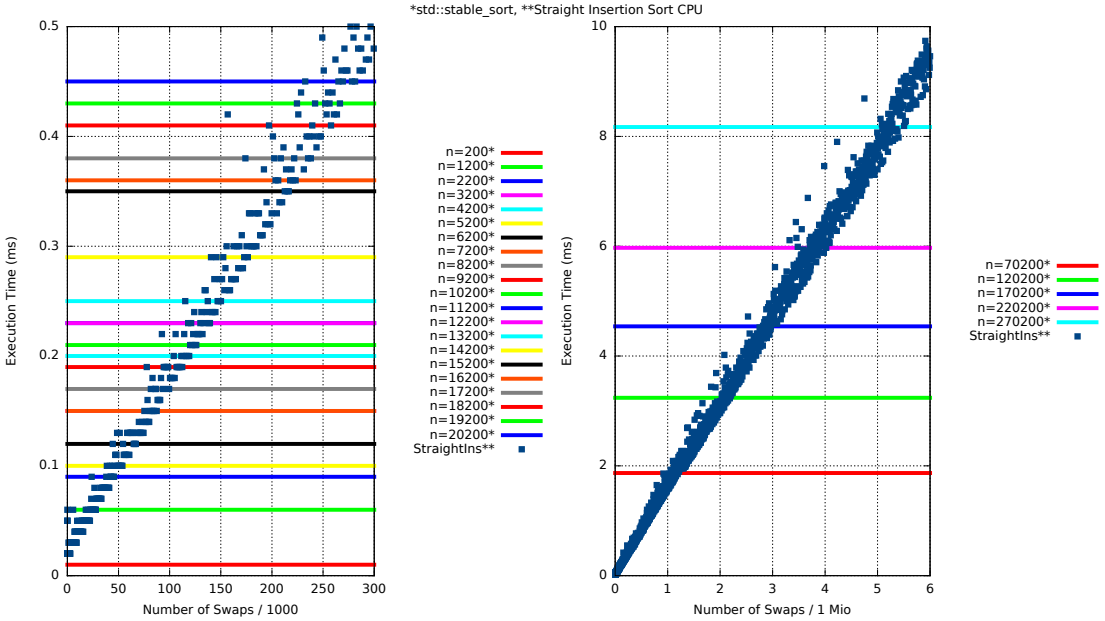

Figure 6.3 CPU in-depth analysis. Tested with OS: Ubuntu 12.04 i386, CPU: Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz, Memory: 4 GB.

The figure's left side shows input lengths up to 20,200 elements, which should refer to most "small size" 3D computer graphic scenarios. The figure's right side shows input lengths up to 270,200 elements, which should refer to bigger scenarios or other applications than collision detection. As a result, the occuring intersections between timings of *Straight Insertion Sort* and `std::stable_sort` can be found in table 6.6.

Input Length	# Swaps	Time (ms)	Input Length	# Swaps	Time (ms)
200	0	0.01	13,200	145,000	0.25
1,200	25,000	0.06	14,200	150,000	0.29
2,200	47,000	0.09	15,200	200,000	0.35
3,200	90,000	0.15	16,200	210,000	0.36
4,200	100,000	0.2	17,200	225,000	0.38
5,200	50,000	0.1	18,200	250,000	0.41
6,200	60,000	0.12	19,200	260,000	0.43
7,200	90,000	0.15	20,200	270,000	0.45
8,200	95,000	0.17	70,200	1,000,000	1.9
9,200	97,000	0.19	120,200	1,900,000	3.2
10,200	100,000	0.21	170,200	2,800,000	4.5
11,200	140,000	0.23	220,200	3,800,000	6.0
12,200	140,000	0.23	270,200	5,000,000	8.2

Table 6.6 CPU in-depth analysis: Execution Time Intersections between *Straight Insertion Sort* and `std::stable_sort`. Notice: The number of swaps are rough values.

6.5.5 Improvement: Individual InvThresholds CPU

Analyzing the data, it is possible to determine good parameter values for making *Adaptive-FrameSort* adaptive for different input lengths. Consider table 6.8 for closer understanding of its working principle. These values were successively gained by decreasing the number of swaps per interval and measuring the resulting timings for the clothball scenario, as can be seen in table 6.7. The chosen parameter for input lenghts below 4,000 is 10,000 number of swaps. Regarding the table it has the highest number of winning frames. At the same time it is not wise to take smaller thresholds as the size of resulting sort ranges might decrease rapidly (especially in random other scenarios). For the chosen parameters' efficiency relate to table 6.8.

NoS for n<4,000[1]	No. of relevant frames	No. of winning frames	nlogn time (ms)[2]	Adaptive Time (ms)[3]
static 60,375*	5 / 94	3	825	821
40,000	86 / 94	7	769	815
20,000	66 / 94	7	782	796
10,000	34 / 94	9	818	796

Table 6.7 Individual number of swaps thresholds for CPU: Analysis based on the *clothball* scenario. *Static number of swaps threshold for all input lengths (old algorithm), [1]NoS threshold for input length of <4,000, [2]Total nlogn algorithm time (ms), [3]Total *AdaptiveFrameSort* time

Inp. Length (up to)[1]	NoS (up to)[2]	Part of int. val.[3]	BC-Time (ms)[4]	WC-Time (ms)[5]
4,000	10,000	10%	0.04	0.24
13,000	20,000	13.8%	0.05	0.3
20,000	37,500	13.8%	0.08	0.53
120,000	75,000	3.9%	0.14	3.34
170,000	350,000	12.5%	0.56	5.06
220,000	550,000	14.5%	0.91	6.91
∞	750,000	<19.7%	1.21	>7.21

Table 6.8 Individual number of swaps thresholds for CPU. [1]Input Length, [2]Number of Swaps, [3]Number of swaps chosen as parameter compared to the intersecting value from table 6.6, [4]Time in Best Case: n-nsquare algorithm succeeds, [5]Time in Worst Case: Number of swaps exceeds efficient time for n-nsquare algorithm; fallback to nlogn algorithm. This time is the sum of "Time in Best Case" plus the time that the nlogn algorithm takes for sorting

You can see that efficient number of swaps thresholds of around 13% of their intersecting values from table 6.6 are chosen. Note that the exception of around 4% for an input length of 120,000 is used to compensate the gap to the previous step of an input length of only 20,000. The gap is that huge because most scenarios would refer to input lengths below 20,000 or starting from 120,000 in the presented 3D scenarios. Gradual settings for input lengths will be more valid for general scenarios.
At this point, a good approach for improving the algorithm might be to include a method

that calibrates the system that the code is running on. Having around 1,000 iterations for each algorithm and taking around 13% of their intersecting number of swaps as the algorithm's parameters, the codes yield should be optimal.

6.5.6 In-Depth Algorithm Comparison on GPU with CUDA Compute Compatibility 1.3

The GPU version of *AdaptiveFrameSort* uses a self-written implementation of *Cocktailsort* as n-nsquare algorithm. *Cocktailsort* in general has a best case complexity of O(1) (movements) and O(n) (comparisons) and a worst case complexity of O(n^2) (see "Bubblesort"[OW12]). The algorithm is an improved version of *Bubblesort*; the advantage is that in an ascending list small elements from the end of the list move as fast to the beginning as big elements from the beginning to the end [Big+08]. For a general tendency comparison between *Bubblesort* and *Cocktailsort* refer to figure 6.1.
For non-atomic data types, `thrust::stable_sort` makes use of merge sort with a complexity of O($n \cdot \log n$)[SHG09].
Equivalent to section 6.5.4, also the GPU algorithms have been measured for different input lengths for `thrust::stable_sort` and different number of swaps for *Cocktailsort*, as can be seen in figure 6.4.

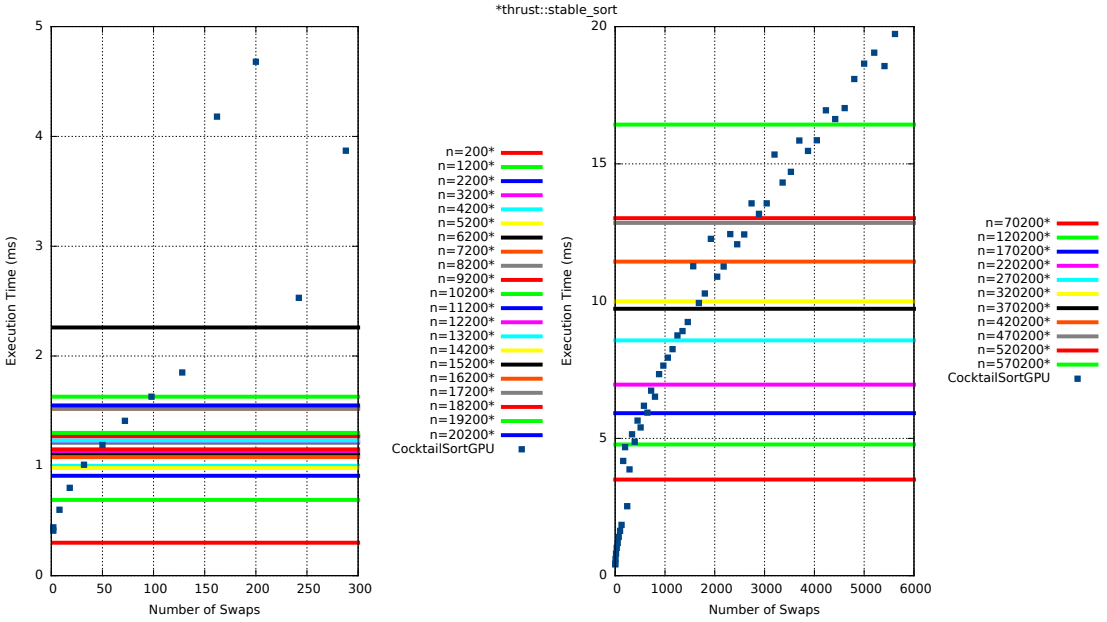

Figure 6.4 GPU in-depth analysis with CUDA Compute Compatibility 1.3. Tested with OS: Ubuntu 12.04 i386, CPU: Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz, Memory: 4 GB, Graphics card: NVIDIA GeForce GTX 260, NVIDIA Driver: 331.62.

Again, on the left side can be seen input lengths from 200 to 20,200, whereas on the right side input lengths from 70,200 to 570,200 are prevailing.
Comparing figure 6.3 and 6.4, two measurements are directly eye-catching. *Straight Insertion Sort*'s execution time is directly linear to the number of swaps needed. *Cocktailsort*'s execution time is rather super linearly dependent on the number of swaps, with a logarithmic-like start. This indicates that the implemented algorithm should be still improved.

As second impression we can see that the number of swaps that result in a faster adaptive sort than with the respective $O(n \cdot \log n)$ algorithm is very low on GPU compared to the CPU version. For an input length of 20,200 elements, *Straight Insertion Sort* could have up to 270,000 necessary swaps to compete with `std::stable_sort`. On GPU, for the same input length already 200 (!) number of swaps will exceed the execution time of `thrust::stable_sort`. Hence, the GPU version of *AdaptiveFrameSort* will be decisively less efficient compared to the CPU version.

Input Length	# Swaps	Time (ms)	Input Length	# Swaps	Time (ms)
200	-*	0.3	170,200	512	5.92
1,200	8	0.69	220,200	800	6.96
2,200	18	0.91	270,200	1,152	8.57
3,200	32	1.13	320,200	1,458	9.99
8,200	50	1.21	420,200	2,178	11.44
20,200	72	1.55	470,200	2,592	12.85
70,200	128	3.5	570,200	4,050	16.43
120,200	200	4.78			

Table 6.9 GPU in-depth analysis with CUDA Compute Compatibility 1.3: Execution Time Intersections between *Cocktailsort* and `thrust::stable_sort`. Notice: The number of swaps are rough values. This table was reduced to pairs that make use of a different number of swaps. *For input lengths around 200 elements the overhead for *Cocktailsort* is higher than for `thrust::stable_sort`.

6.5.7 Improvement: Individual InvThresholds GPU

Taking the data from table 6.9, it is obvious that a clothball-animation based parameter adjustment will not be profitable. Hence I decided to set the parameters manually the following generous way, as can be found in table 6.10. Generous in this context means that the bigger the number of swaps per input length is, the bigger the resulting size of the corresponding sort range will be. Bigger n-nsquare sort ranges can result in much faster adaptive sorts for the whole input data. At the same time though, the saved time per sort range will become smaller. It is to find a best general value in this balance.
As a GPU specific feature, note the following: The adaptive times do not reflect the real computation time for each sort range but the complete computation time for ALL n-nsquare sort ranges, since their execution will be in parallel. That means: The more n-nsquare sort ranges exist, the faster is the adaptive sorting there, but also the longer the range merge will take afterwards.

Inp. Length (up to)[1]	NoS (up to)[2]	Part of int. val.[3]	BC-Time (ms)[4]	WC-Time (ms)[5]
1,200	0	0%	-	0.69
2,200	8	44.4%	0.6	1.51
11,200	18	36.0%	0.8	2.07
20,200	26	36.1%	0.9	2.45
220,200	240	30.0%	2.53	9.49
520,200	1,000	38.5%	7.75	20.77
∞	1,500	<37.0%	10.24	>23.26

Table 6.10 Individual number of swaps thresholds for GPU with CUDA Compute Compatibility 1.3. [1]Input Length, [2]Number of Swaps, [3]Number of swaps chosen as parameter compared to the intersecting value from table 6.6, [4]Time in Best Case: n-nsquare algorithm succeeds, [5]Time in Worst Case: Number of swaps exceeds efficient time for n-nsquare algorithm; fallback to nlogn algorithm. This time is the sum of "Time in Best Case" plus the time that the nlogn algorithm takes for sorting

6.5.8 In-Depth Algorithm Comparison on GPU with CUDA Compute Compatibility 3.5

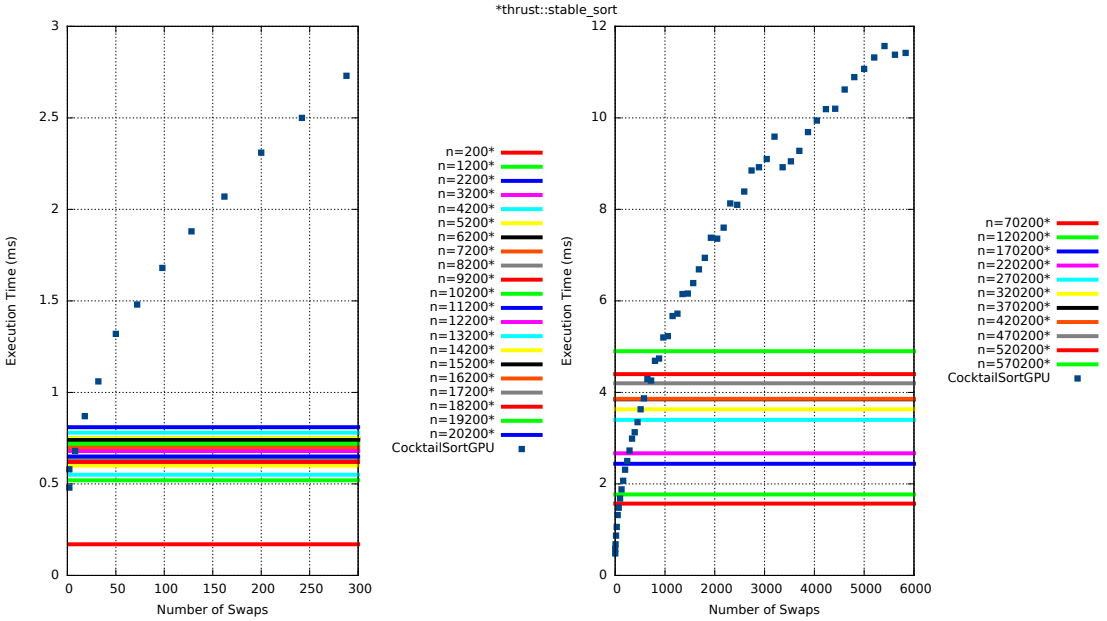

Figure 6.5 GPU in-depth analysis with CUDA Compute Compatibility 3.5. Tested with OS: Windows 7 Enterprise 64bit SP1, CPU: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz, Memory: 16.0 GB Ram, Graphics card: NVIDIA GeForce GTX 780, NVIDIA Driver: 340.52.

For Kepler architecture based graphics cards it turns out that the Thrust library is already very highly optimized. Compared to the Tesla architecture based results found in figure 6.4, it seems very hard to compete with the non-adaptive algorithm. Based on the same approach from last chapter, the resulting individual thresholds hold for Kepler architecture graphics cards can be found in table 6.11.

Inp. Length (up to)[1]	NoS (up to)[2]	Part of int. val.[3]	BC-Time (ms)[4]	WC-Time (ms)[5]
20,200	0	0%	-	0.81
70,200	8	11.1%	0.68	2.25
270,200	50	11.1%	1.32	4.72
∞	98	<11.1%	1.68	>6.58

Table 6.11 Individual number of swaps thresholds for GPU with CUDA Compute Compatibility 3.5. [1]Input Length, [2]Number of Swaps, [3]Number of swaps chosen as parameter compared to the intersecting value from table 6.6, [4]Time in Best Case: n-nsquare algorithm succeeds, [5]Time in Worst Case: Number of swaps exceeds efficient time for n-nsquare algorithm; fallback to nlogn algorithm. This time is the sum of "Time in Best Case" plus the time that the nlogn algorithm takes for sorting

6.6 Improvement: Fast Sort Range Computation

The last improvements in the field of sort ranges in this work were to have a fixed number of ranges set up for any kind of scene (see chapter 6.4). This means, that only the resulting length for ranges in different scenes vary. The task for *AdaptiveFrameSort* is to determine the sum of all the items' sort distances within each range.

There are different ways to solve this problem. In general, you can refer it as "Prefix-Sum" problem. The Thrust library, for example, offers `thrust::inclusive_scan` as option for building prefix-sums. Alternatively, you might also use `thrust::reduce` to get a sum over the items as reduction. All possible algorithms were implemented and timings for different numbers of ranges compared in table 6.12.

# Ranges	CPU	thrust::inclusive_scan*	thrust::reduce*	Own Implementation*
20	5 ms	4 ms	2 ms	1 ms
40	5 ms	4 ms	3 ms	1 ms
80	5 ms	4 ms	5 ms	1 ms

Table 6.12 Time comparison of different algorithms for calculating the range sums within the clothcar scene for frame 0. *Tested with CUDA compute compatibility 1.3

There are different results to read out of this table. First of all, the CPU version, `thrust::inclusive_scan` and the own implementation are stable, which means, they are (practically)

independent on the the number of ranges.
As for `thrust::reduce`, this algorithm is called for each range to determine its sort distance sum. This, of course, generates an overhead for calling the function kernels each time.
Overall, for the CPU it takes around 5 ms to sum up all distances in the array of length 1,384,312.
The next comparison is decisive and explains why there is a need for a self-written implementation of calculating the range sums. One would assume that `thrust::inclusive_scan` behaves fastest in this context. Although after execution another kernel has to be run in order to compute each range's individual sum (which computes roughly by

$$\begin{aligned} rangeSum_i = & sortDistance(inputElement_{(i+1)*minimumRangeLength-1}) - \\ & sortDistance(inputElement_{i*minimumRangeLength-1}) \end{aligned}$$

and in tests this kernel's execution time was around 0 ms, so not much worth to mention), the use of `thrust::inclusive_scan` will be the fastest in comparison; for summing up all item's sorts distances, it will be hard to compete with the optimized Thrust algorithm. Nevertheless, two reasons speak against using this version: *AdaptiveFrameSort* takes the range sums to compute sort ranges. A sort range can have the maximum size of the whole input array's length. A sort range will be of extraordinary use, if its distance sum does not exceed the individual InvThreshold for its length (because then the n-nsquare algorithm can be used). Looking into tables 6.8 and 6.10 for a length of the clothcar's bounding boxes, we receive maximum thresholds of 750,000 for CPU and 1,500 for GPU (CUDA compute compatibility 1.3). While calculating the range sums, these thresholds can be used already. If the sum for a single range exceeds the overall threshold, it can stop any further calculation and simply return a value that is bigger than this threshold. Table 6.13 clarifies what happens.

Range	Distance Sum (actual)	Distance Sum (CPU)	Distance Sum (CUDA 1.3)
0-6	0	0	0
7	96,320,655	754,526	1,501
8	245,959,607	751,045	1,501
9	514,295,667	750,078	1,501
10	623,794,983	761,010	1,501
11	397,933,947	757,251	1,501
12	163,537,241	752,391	1,501
13	66,531,532	751,366	1,501
14-19	0	0	0

Table 6.13 Range sums calculated by the own sum implementation for clothball frame 40

The table reveals a huge difference in actual sums and sums after a fallback. Whereas the sum for range 9 is actually over 514 million big, the actual usable value is already 750,078 for CPU and 1,501 for GPU. This means that the sum algorithm can stop already very early while computing the sums. Thrust's implementation cannot be told such kind of fallback.

Another and even more severe reason for using a self-written implementation for calculating the range sums instead of Thrust's algorithms is that of a data type issue. Although the used Thrust algorithms are fully template based and theoretically a `long long int` data type could probably hold any realistic data ranges, performance-wise it is inefficient to introduce additional payload for each sorting element, in case it can be avoided. Section 7.2 will demonstrate differences between using more memory space for each element in an array and using less memory, where, as can be assumed, bigger data structures for each element result in longer sorting times. So, in order to save space, the variable holding each item's distance sum is of type `int`. The maximum positive number that the `int` type can hold is (in a 4-byte implementation, may vary on different systems) $2^{31} - 1 = 2,147,483,647$. The clothcar scene does not exceed this limit with any range sum, though there might be scenes whose range sums exceed this value easily. The for *AdaptiveFrameSort* usable data range however only needs to cover values up to the individual InvThreshold; any range sum calculation beyond this value does not represent any useful information to the algorithm. Practically though, there is no way of limiting sum computations with Thrust's algorithms. Concluding, the use of a self-written implementation is welcome and also necessary; and its timings beat all other implementations (a constant 1 ms, range number independent). Note, that in case of the GPU version of *AdaptiveFrameSort*, the computation of

range sums happens fully on GPU, but the length calculation for each sort range still happens on CPU (also in the final version). The computation is trivial, but a small but negligible overhead appears by copying the range sum values (which in the current configuration is an array with 20 integer values) from VRAM to RAM. Since the computation is not parallelizable efficiently and it only involves small adjustments of boundaries, it represents a justifiably small overhead. It is important to note that in any case the computed sort ranges would have to be copied to the CPU RAM in order to manage the algorithm's general proceedings. In case of the clothcar scene, instead of 20 values of type `int`, three sort ranges would have to be copied, where a `SortRange` is a data structure consisting of two `int` values and one `bool` value. In the specific example run on the used architecture, instead of 27 bytes for the sort ranges, 80 bytes for the range sums are copied from VRAM to RAM. Therefore, in terms of memory copying times, no important overhead is introduced here.

6.7 Improvement: Block Based Parallel N-Nsquare Sort

The implemented *Cocktailsort* algorithm only uses one CUDA block for sorting. Since often there will be more than one n-nsquare sort range, it will be a good idea to sort these ranges each in a separate CUDA block, launching them in the same kernel call. The sorting of the two n-nsquare sort ranges in the clothcar secene (frame 9) takes 2 ms for the first range and 1.3 ms for the second range. Sequentially, the total execution time for these two ranges then is around 3.3 ms. If they are run together, the total execution time will be around 2 ms only. Figure 6.6 clarifies the outcome. Overall, for the clothcar scene this improvement means less "jumpy" execution times for several frames.

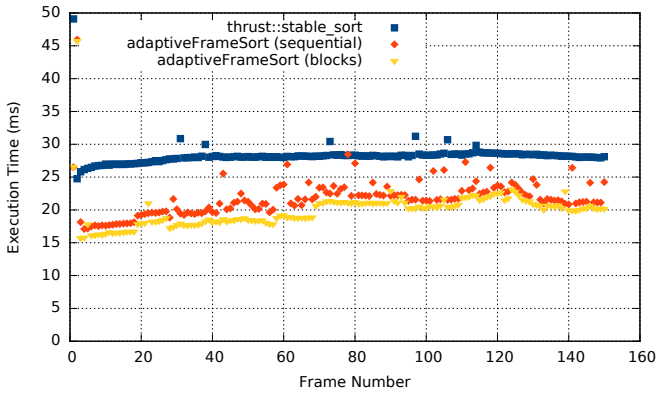

Figure 6.6 Comparison of sequential n-nsquare range sort and block based parallel n-nsquare range execution for the clothcar scene (CUDA compute compatibility 1.3)

On the other hand, it would also not make sense to sort more than one sort range in one block, since n-nsquare ranges often are separated by nlogn ranges (as, for example, in the clothcar scene) and do not represent one coherent sorting array, which would complicate data structures in the sorting kernel unnecessarily and the amount of available shared memory even on Kepler architecture based graphics cards is not sufficient to hold the data already for a single bigger n-nsquare range.

6.8 Improvement: Merging Consecutive nlogn Subsets After n-nsquare Fallbacks

This improvement is not as straight forward as the previous one. Sorting subsets of data and afterwards merging them is an approach for a diverse number of sorting methods. In the case of nlogn sorting algorithms, the following table can clarify the ideas behind it. Assuming we have a data array with 100 elements and want to subdivide it equally sized subsets that will get merged afterwards, we might have a scenario as in table 6.14.

Subsets	Size of Each Subset	Formula	Theoretical Runtime
1	100	$1 \cdot 100 \cdot \log 100$	200
2	50	$2 \cdot 50 \cdot \log 50$	169
4	25	$4 \cdot 25 \cdot \log 25$	139
50	2	$50 \cdot 2 \cdot \log 2$	30

Table 6.14 $n \cdot \log n$ subdivision example for n=100

The original runtime complexity for sorting an array with a length of 100 in one pass is

$$Runtime = 100 \cdot \log 100 = 200$$

For the best case of subdivision (50 separately sorted subsets of length 2) let us assume two different data related scenarios. One is where after sorting all subsets the whole array with length 100 is already sorted. This happens, if the original data was relatively pre-sorted. In this case let us assume an adaptive in-place merge algorithm has the complexity O(1) (through only comparing the rightmost element of the left set and the leftmost element of the right set). Then

the runtime for sorting all subsets and merging them afterwards will be:

$$Runtime = mergeTime + 50 \cdot 2 \cdot \log 2 = 79$$
$$mergeTime = 49 \cdot 1$$

Now the worst case for the merge might be that all the data from the first subset has to be actually put into the range of the last subset and so on. It is sorted inversely (based on the first element of each subset). In this case, a merge would cost O(n-1) many steps:

$$Runtime = mergeTime + 50 \cdot 2 \cdot \log 2 = 2,529$$
$$mergeTime = 3 + 5 + \ldots + 99$$
$$mergeTime = \frac{n^2}{4} - 1 = \frac{100^2}{4} - 1 = 2,499$$

In this sense, it can be worth dividing a sort into several subsets and merge them afterwards (79 steps compared to 200 original steps) but it can also cost a lot of additional time (2,529 steps). In the case of frame based collision detection this case would probably rather not happen (in a way that one object moves from the right side of the scene to the left side from one frame to the next), but suddenly fast movement cannot be excluded. In addition, the calculation example is only on a theoretical base; in reality algorithms also have some initial overhead, for example.

In case of the GPU version, merging consecutive nlogn sort ranges at any time into one (when finding the range and after the fallback of an n-nsquare sort range) is the right way to go. In a comparison run with 100 iterations for frame 2 of the clothball scene, *AdaptiveFrameSort* had an average time of 5.3 ms with merging the fallback sort ranges and 11.7 ms without merging them. CUDA kernel calls simply have too much overhead and also the implemented non-adaptive merge of the sort ranges consumes to much time for a higher amount of sort ranges.

In case of the CPU version, no merge of nlogn sort ranges is actually faster for two of the three researched scenes. Figure 6.7 shows a direct comparison.

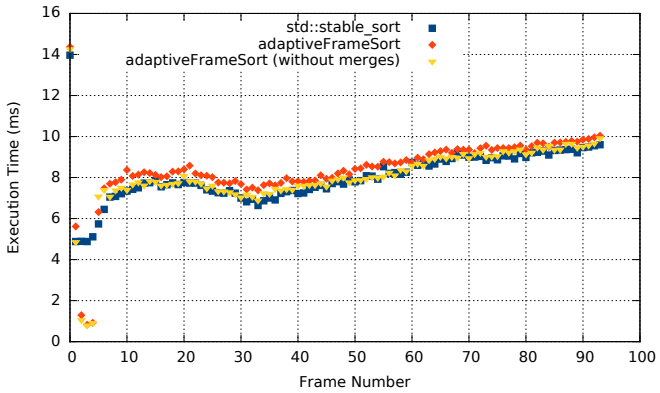

Figure 6.7 Time comparison for the clothball scene, with and without merging any nlogn sort ranges

The algorithm without merges is around 33 ms faster for sorting all frames than the algorithm with merges. In the funnel scene, it is around 16 ms faster. The timing curve also seems to be visually closer to the original `std::stable_sort` algorithm.

Nevertheless, in the clothcar scene for example, the algorithm is around 272 ms slower (time win after all frames in comparison with `std::stable_sort`, without merging: 9,822 ms, with merging: 10,094 ms). On CPU, this means, that no merge can be faster (with up to 20 sort ranges in the current configuration, which is the amount of "ranges") but also slower through too difficult merges. On GPU the merges are faster in any case. The final implementation of the algorithm will merge the nlogn sort ranges both for searching and after n-nsquare fallbacks for the GPU version and the CPU version alike. This way, the algorithm is valid for more general applications and scenes could also contain faster moving segments.

6.9 The Final Algorithm: AdaptiveFrameSort

This paragraph will show the approach of the final version of *AdaptiveFrameSort*, as it is referred to throughout this work. Taking a practical example of one dimensional collision detection with AABBs, it will explain in detail which steps the algorithms takes. In section 6.9.2, the pseudocode will reveal closer functionality.

6.9.1 Functionality Explained by an Example

In order to clarify the functionality of *AdaptiveFrameSort*, consider figure 6.8.

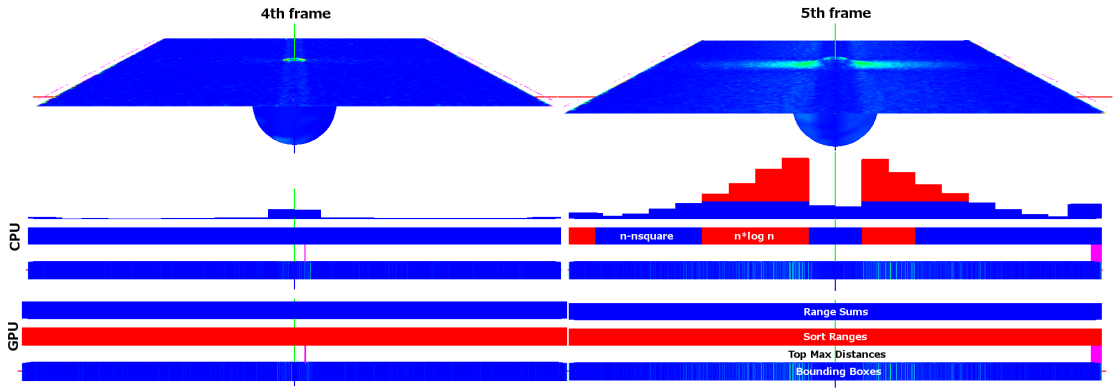

Figure 6.8 Detailed InvThreshold results for the clothball scene. Number of "Range Sum" ranges: 20. Width of one "Range Sum" range: 9,223. The "Range Sum" threshold is set to a constant value of 60,375 (CPU) and 250 (GPU) (beyond this, the "Range Sums" are marked red instead of blue)

The clothball scene features a representative example for the algorithm's approach. You can see two screenshots for frames 4 and 5 and respectively 4 bars for both the CPU and GPU algorithm. The respectively lowest bar visualizes the frame's bounding boxes. The colors used in this bar and in the objects' screenshots are false colors as explained in section 5.2. The "Top Max Distances" bar shows all the bounding boxes in pink that moved along a maximum distance after being sorted in the last frame. All triangles that have this max distance (or at least 80% of the max distance) are marked the same way in the object's screenshot (in thise case the cloth's edges move at fastest speed compared other parts of the cloth and result in most X-axis movement). The length of each "Top Max Distance" bounding box implies a good estimation about the length of its sorted distance compared to the overall frame. Technically, its extent is its distance plus and minus around its position. That means, it actually shows double of its distance. This way though, it is easier to figure out where it "came from".

Before explaining the "Sort Ranges", it is easier to clarify the function of the "Range Sums". Relative to the respective InvThreshold (which is 60,375 for CPU and 250 for GPU) the height of each range sum (there are 20 altogether) indicates the number of unsorted pairs in this region. From the range sums it is easy to see the section where the cloth hits the ball: first in the middle (frame 4) and then with the parts next to the middle (frame 5). Remember that the range sums indicate movement. Once the cloth has hit the top of the ball, it will not move there anymore, it has collided. The cloth's corners result in blue range sums. This means that their number of unsorted pairs has not exceeded the respective threshold and this range might possibly be sorted by the fast n-nsquare algorithm. The parts around the middle in frame 5 exceed the threshold and the bars continue in a red color beyond the threshold. These ranges could most likely not be sorted by the fast n-nsquare algorithm, since the nlogn algorithm would be faster there. Note that since the threshold for the GPU version is very low, relative heights will result in very high

bars in most cases, so there are only blue bars visible (which means nevertheless that the range sums are exceeding the corresponding thresholds in frames 4 and 5).

At first glance, the sort ranges do not evolve as obviously as the other datasets. Talking about the CPU version, the actual number of sort ranges is not six as can be seen in the figure but eight. This is based on the fact that consecutive nlogn ranges get merged to one range but consecutive n-nsquare ranges do not. An explanation for that will follow up later in this chapter.
The first range sum is marked blue for frame 5, nevertheless the corresponding sort range beneath is marked red. At this point, note that the final implementation has a dynamic distance sum threshold value; the visualized range sums use a constant threshold value, though (which is based on the old algorithm's approach, but fits for visualization). The calculation of range sums has become input-length depend since the introduction of individual InvThresholds. One range sum has the length of 9,223 elements in case of the cloth scene (for 20 range sums altogether). Regarding table 6.7, the individual InvThreshold for an input length of 9,223 is 20,000. The first range's sum of unsorted pairs is 20,738, though. Hence, the algorithm chooses this range to be sorted by an nlogn sorting algorithm. The next four ranges have the sums 8,982; 18,158; 34,976; 54,712. Their sum is 116,828. Four ranges together make an input length of $4 \cdot 9,223 = 36,892$. This input length has a respective individual InvThreshold of 75,000. This means, that ranges' unsorted pair sums exceed the threshold (116,828 > 75,000). At this point, the algorithm jumps one range back. The length for three ranges is 27,669 and the corresponding InvThreshold still is 75,000. The three ranges' unsorted pair sum 62,116, which is smaller than 75,000. Hence, the next n-nsquare sort range will consist of the ranges 2-4. With the same kind of fallback and — dependant on the current input length — individual InvThresholds, all other sort ranges are computed. For example, the next range, range 5, will be also counted as n-nsquare sort range. Ranges 6-9 will be one a nlogn algorithm sort range.

It is important to note that consecutive nlogn sort ranges are being merged to one sort range, whereas consecutive n-nsquare sort ranges (those who separately do not exceed the corresponding InvThreshold) will NOT be merged to one sort range. The reason is simple and useful at the same time: The separete n-nsquare sort ranges will be definitely sorted faster than a nlogn algorithm would take for doing that. After sorting they will be actually merged by a merge algorithm. This is faster, than sorting the ranges together by one n-nsquare algorithm without having to merge them afterwards, as for the obvious reason that an n-nsquare algorithm grows linearly to the number of swaps, whereas `std::inplace_merge` grows linearly to the number of elements that need to be merged [SGI14a] (which is faster in most cases).

6.9.2 Pseudocode

This section contains the pseudocode of *AdaptiveFrameSort*. Note that the pseudocode is highly simplified and only referring to the CPU version of the algorithm; inside the working and optimized C++ implementation, it contains additional code in order check boundaries and to provide GPU support as well. In addition, the code of the underlying n-nsquare and nlogn algorithms is missing. Refer to chapter 6.5.3, in order to see which algorithms are used for which sort range type.

As the method `adaptiveFrameSort` shows, the algorithm's major steps are: Sort sort ranges based on the last frame, merge them, compute distances and range sums and build new sort ranges for the next frame. The function `getIndividualInvThresholdByLength` refers to individual InvThresholds as described in the respective first two columns of the tables 6.8, 6.10 and 6.11. For example, in case a sort range has a length of 11,942 on CPU, the function returns an individual InvThreshold of 20,000 (refer to table 6.8). This means, that in case the number of swaps necessary to sort the sort range in the last frame was less than 20,000, in the current frame this sort range will be sorted by the n-nsquare algorithm corresponding to table 6.4. Otherwise, it will be sorted by the corresponding nlogn algorithm.

Listing 6.1 *AdaptiveFrameSort* pseudocode for the CPU version

```
1   // Entry point for each frame sort in an animation
2   // data : array of struct that contains integer
3   //        sortPositionAndAfterSortDistance and is to be sorted
4   // previousFrame : previous frame's adaptiveFrameSort return value
5   //                 containing calculated sort ranges and the
6   //                 max sort distance
7   // numberOfRanges : desired number of ranges to compute sort
8   //                  distance sums of; the resulting number may
9   //                  be smaller based on a minimum range length
10  // returns frame information for use in next frame's sort
11  // after execution, data contains sorted elements
12  procedure adaptiveFrameSort(data, previousFrame, numberOfRanges)
13      // Prepare sort positions
14      prepareSortPositions(data)
15      // 1. Sort subsets based on sort ranges from previous frame
16      sortSubsets(previousFrame.sortRanges)
17      // 2. Merge
18      mergeAll(data, previousFrame.sortRanges, previousFrame.maxDistance)
19      // 3. Compute distances
20      frame.maxDistance = computeDistancesAndGetMaxDistance(data)
21      // 4. Compute range sums
22      rangeSums = computeRangeSums(data, numberOfRanges)
23      // 5. Compute sort ranges for next frame based on range sums
```

```
24          frame.sortRanges = computeSortRanges(rangeSums)
25
26          return frame
27      end procedure
28
29      // Set sort positions to their current position,
30      // meaning a distance of zero, so that
31      // abs(i - data[i].sortPositionAndAfterSortDistance) == 0
32      procedure prepareSortPositions(data)
33          for i = 0 to data.length-1 inclusive do
34             data[i].sortPositionAndAfterSortDistance = i
35          end for
36      end procedure
37
38      // Sort n-nsquare sort ranges, merge fallback ranges with nlogn ranges
39      // and sort all nlogn ranges
40      // nnsqSortAllRangesIn returns true, if any n-nsquare sort had
41      // a fallback
42      // nnsqType returns all sort ranges of type n-nsquare
43      // nlognType returns all sort ranges of type nlogn
44      procedure sortSubsets(sortRanges)
45          nnsqFallback = nnsqSortAllRangesIn(nnsqType(sortRanges))
46          if nnsqFallback then
47              mergeConsecutiveNlognRanges(sortRanges)
48          end if
49          nlognSortAllRangesIn(nlognType(sortRanges))
50      end procedure
51
52      // For three or less sort ranges, try to merge only borders (heuristic)
53      // Otherwise or if the array is still unsorted, merge all sort ranges
54      // consecutively
55      procedure mergeAll(data, sortRanges, maxDistance)
56          needsGlobalMerge = sortRanges.length > 3
57          if sortRanges.length > 1 then
58              if not needsGlobalMerge then
59                  mergeLocalHeuristic(data, sortRanges, maxDistance)
60                  needsGlobalMerge = not isSorted(data)
61              end if
62              if needsGlobalMerge then
63                  mergeGlobally(data, sortRanges)
64              end if
65          end if
66      end procedure
```

```
67
68  // Try to merge small ranges around the borders of existing sort ranges
69  // with a size of two times the last maximum global sort distance
70  // for an element
71  procedure mergeLocalHeuristic(data, sortRanges, maxDistance)
72      i = 1
73      while i < sortRanges.length do
74          borderMergeRange.start = sortRanges[i].start -
75              2*maxDistance
76          borderMergeRange.end = sortRanges[i].start +
77              2*maxDistance
78          merge(data, borderMergeRange.start,
79                      sortRanges[i].start,
80                      borderMergeRange.end+1
81               )
82          insertRangeInto(borderMergeRange, sortRanges)
83      end while
84  end procedure
85
86  // Merge all sort ranges consecutively; in case of a CPU merge,
87  // this happens adaptively
88  procedure mergeGlobally(data, sortRanges)
89      for i=1 to sortRanges.length-1 inclusive do
90          merge(data, 0, sortRanges[i].start,
91                         sortRanges[i].end+1)
92      end for
93  end procedure
94
95  // Compute the sort distance by substracting each item's sorted position
96  // from its original position
97  // returns the global max sort distance
98  procedure computeDistancesAndGetMaxDistance(data)
99      maxDistance = 0
100     for i = 0 to data.length-1 inclusive do
101         data[i].sortPositionAndAfterSortDistance =
102             abs(i - data[i].sortPositionAndAfterSortDistance)
103
104         if data[i].sortPositionAndAfterSortDistance > maxDistance then
105             maxDistance = data[i].sortPositionAndAfterSortDistance
106         end if
107     end for
108     return maxDistance
109 end procedure
```

```
110
111     // Compute sort ranges for the next frame based on range sums,
112     // configured by an individual invThreshold depending on the length of
113     // each potential sort range
114     // returns sort ranges resulting from this frame
115     procedure computeSortRanges(rangeSums)
116         sortRanges = []
117         rangeNumber = 0
118         sortRange.end = -1
119         while rangeNumber < rangeSums.length do
120             sortRange.start = sortRange.end+1
121
122             distanceSum = 0
123             do
124                 distanceSum = distanceSum + rangeSums[rangeNumber]
125                 rangeNumber = rangeNumber + 1
126                 individualInvThreshold = getIndividualInvThresholdByLength(
                        rangeNumber*minimumRangeLength)
127             while distanceSum < individualInvThreshold
128               and rangeNumber < numberOfRanges
129             end do
130
131             sortRange.isNnsqRange = distanceSum < individualInvThreshold
132             sortRange.end = rangeNumber*minimumRangeLength-1
133
134             sortRanges.add(sortRange)
135         end while
136
137         return sortRanges
138     end procedure
```

Chapter 7

Detailed Results

This chapter will outline detailed results based on the final version of *AdaptiveFrameSort*. It will present thorough performance results for all three animation scenes (chapters 7.1 and 7.2), as well as concluding analyses of the scenes themselves (chapter 7.3). In addition, the performance results will be examined in relation to measures of unsortedness in chapter 7.4. A detailed analysis of the clothcar scene in chapter 7.5 will eventually show up differences between recent technology improvements regarding CUDA.

7.1 Frame Based Timings for All Three Scenes

In terms of collision detection data, *AdaptiveFrameSort* has become an adaptive sorting algorithm. It is an all-rounder — it allows existing CPU algorithms to react adaptively to continuously (slightly) changing data. It also works with existing GPU algorithms which constitutes a central motivation of this work. Nevertheless, this algorithm competes gracefully with the existing adaptive (non-time based) CPU algorithms. Comparing its times to *Timsort*, for example, you find it a bit less fast for the clothball scene in case of rather pre-sorted frames (frames 3-7 in figure 7.1) but also distinctively less slow for rather unsorted frames (after frame 10). The difference sum of execution times (the small graph displayed at the bottom left of the figure) confirms: After 93 frames *AdaptiveFrameSort* is overall 581 ms faster compared to the usage of *Timsort* in every frame.

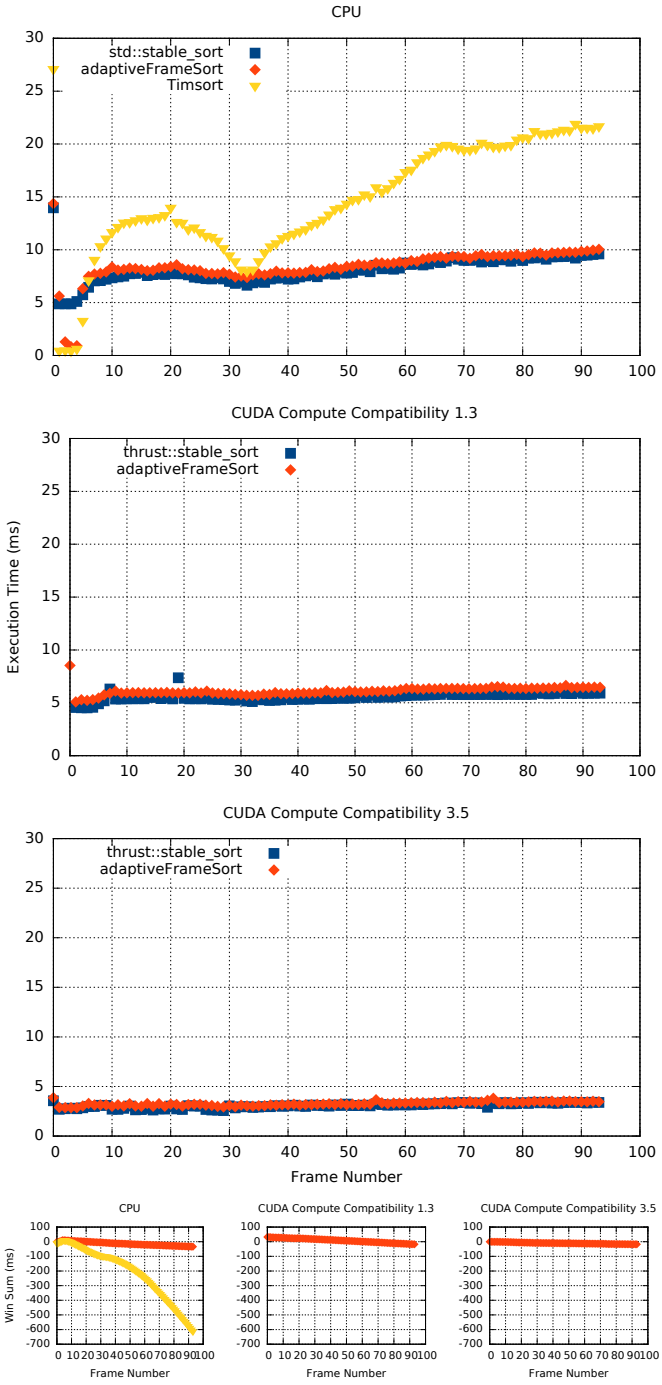

Figure 7.1 Clothball final timing results

The funnel scene in figure 7.2 shows a bit slower code in the rather pre-sorted frames (frames 30-130) but also a distinctively faster code for rather unsorted frames (frames 130-450). The

difference sum, again, reveals that after 500 frames *AdaptiveFrameSort* is overall 452 ms faster than *Timsort*.

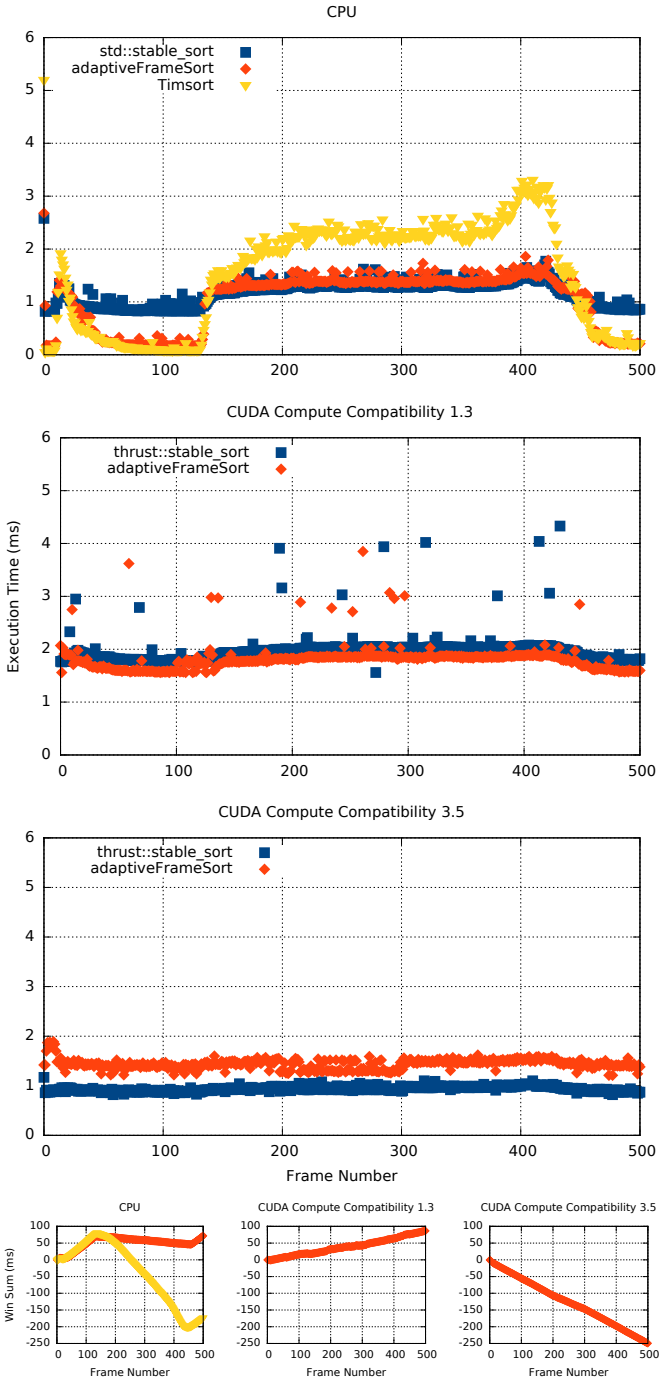

Figure 7.2 Funnel final timing results

In case of adaptive GPU sorting, *AdaptiveFrameSort* can contribute to a novel kind of hybrid GPU algorithms. In general, though, for small input data (<100,000 elements) it is hard to create an efficient adaptive sorting algorithm. The scenes clothball and funnel yield little use for overall adaptive algorithms. After 93 frames of the clothball scene, *AdaptiveFrameSort* is 17 ms slower (CUDA compute compatibility 1.3) than the usage of Thrust's `thrust::stable_sort` in every frame. After 500 frames of the funnel scene, *AdaptiveFrameSort* is 86 ms faster (CUDA compute compatibility 1.3) than the usage of STL's `std::stable_sort` in every frame, though the timing course is similar in both cases which probably leads to a measuring mistake; more cycles than one run should be performed here in order to get more accurate results. CUDA compute compatibility 3.5's results even reveal the downside of *AdaptiveFrameSort*: The overhead is too high compared to Thrust's sorting algorithm. In times of around 1 ms it is impossible to get faster; the overhead will result in bigger execution times in every frame.

The bigger scene, clothcar, though, represents an excellent scenario for adaptive sorting algorithms (see figure 7.3). Also here, *AdaptiveFrameSort* wins overall against *Timsort* as CPU version (after 150 frames 452 ms faster) and also wins against the Thrust's non-adaptive `thrust ::stable_sort` as GPU version. As can be seen in the middle figure, the difference sum of execution times after 150 frames yields 1,227 ms gained in comparison to Thrust's algorithm. This is a lot of time for time critical animations. Unfortunately, this only holds for CUDA compute compatibility 1.3, which stands for Tesla architecture based graphics cards. For CUDA compute compatibility 3.5 (Kepler architecture, right figure) the result is rather dissatisfying. A LOSS of 61 ms compared to Thrust's algorithm is the consequence.

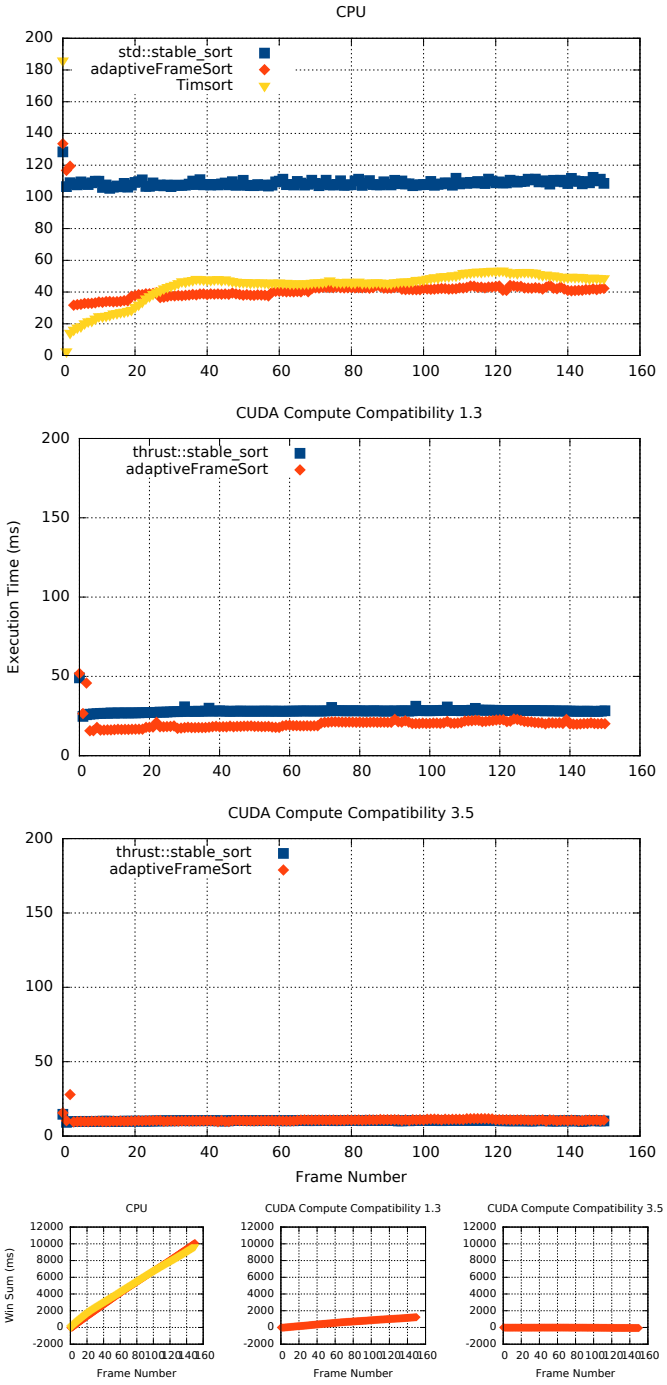

Figure 7.3 Clothcar final timing results

7.2 Frame Based Timing Comparison with Reference to Data Complexity

AdaptiveFrameSort's run complexity is $O(n \cdot logn)$ in worst case and $O(n)$ in best case, both on CPU and GPU (constant summands were neglected here). The algorithm's memory complexity is the following: On the CPU, a constant amount of memory will be allocated for the ranges and sort ranges; this number depends on the settings of number of ranges, which is 20 by default. If the chosen architecture is GPU, the constant memory for a copy of the ranges will be allocated. The chosen architecture's memory (either CPU or GPU) will also hold an additional integer attribute for every element that is supposed to be sorted. In terms of memory complexity, this means that *AdaptiveFrameSort* is in the class of $O(n)$, both for CPU and GPU. An interesting question that arises now, is to compare the previous timing results of *AdaptiveFrameSort* to timing results for the regular algorithms *without* the data overhead of an extra attribute for holding each item's sort distance. This overhead was also measured for these algorithms in the previous analysis, since it was part of the bounding box structure (the attribute is called "sortPositionAndAfterSortDistance", see its definition in section 5.2). Figure 7.4 will give an overview about the differences.

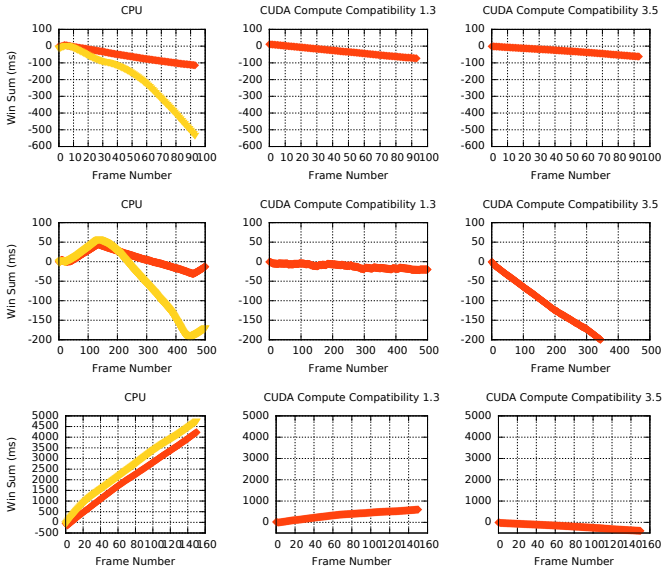

Figure 7.4 Final timing results without AFS data overhead for the algorithms `std::stable_sort` , timsort and `thrust::stable_sort`. From top to bottom: Clothball, funnel and clothcar

Comparing these results to the time wins in figures 7.1, 7.2 and 7.3, it is visible that the overall adaptive result for *AdaptiveFrameSort* stays similar. After 500 frames, the sum for the funnel scene times reaches -12.6 ms, which is a loss, instead of winning 72 ms with the use of data

overhead for the regular sorting algorithms. The win curves remain similar, though; the CPU version still beats *Timsort* in the clothball and the funnel scene. Even the clothcar scene for CUDA compute compatibility 1.3 can record a win of 600.2 ms. It is notable that the bigger data structures get, that are supposed to be sorted adaptively, the less the algorithm's overhead for the additional field of sort distance (of type integer) counts. This is why this additional overhead does not have to be taken into account in general. CUDA compute compatibility 3.5 results stay critical still, though. With a loss of 297 ms after all frames, the win sum especially in the funnel scene is almost 15 times slower compared to CUDA compute compatibility 1.3. In the clothcar scene, the algorithm has an overall loss of around 397 ms. It is to mention, that this loss can be turned into a win, if the cloth had fewer polygons, and especially, if the used *Cocktailsort* implementation was faster with less overhead. In fact, already about half of the computation time is still used for the high fluctuation in the scene's cloth area. For more details refer to chapter 7.5.

7.3 The Scenes' Effectiveness

Analyzing the results, a good question that comes to mind is how effective *AdaptiveFrameSort* actually is. At which frames does a scene animation allow the algorithm to be effective at all? One way of finding out is to have a look at the number of effective *Straight Insertion Sort* sort ranges in the (compared to the GPU version in general more effective) CPU version. An effective sort range is a range that was actually fully sorted by its promoted algorithm. This means that technically the number of effective sort ranges will be determined after the sort ranges were sorted but before they are merged (because this will decrease the number of sort ranges again up to one). Figure 7.5 clarifies this number for each frame of every investigated scene.

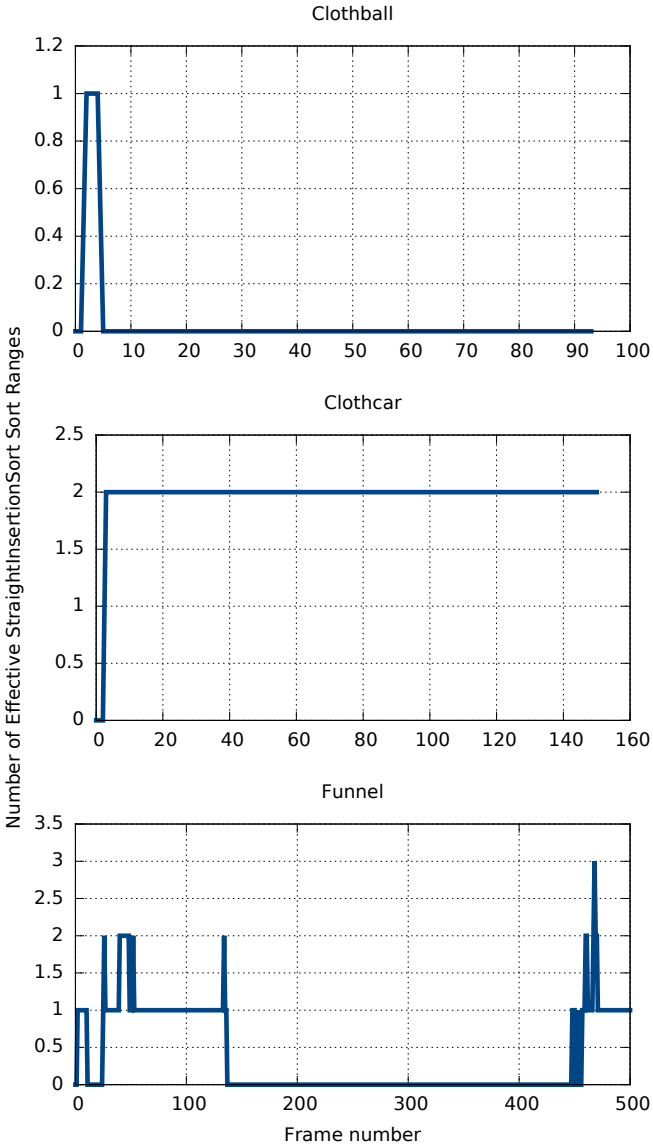

Figure 7.5 *Straight Insertion Sort* effectiveness for the CPU version

Reading these results is more complicated than saying "A higher number of effective *Straight Insertion Sort* sort ranges" results in a faster algorithm. The only measurement that is for sure, is to say that "Zero effective *Straight Insertion Sort* sort ranges do not result in a faster algorithm". This applies very well to the clothball scene: Frame two, three and four are the only frames that allow a partly sort by *Straight Insertion Sort*. In addition, frames three and four are sorted entirely by *Straight Insertion Sort* (this is not visible in the graph here). So these three frames will be sorted faster by *AdaptiveFrameSort* than by `std::stable_sort`. The proof can be

found in figure 7.1.

The clothcar scene is quite stable in terms of pre-sorted data ranges. Starting with frame three, all subsequent frames will be sorted by two *Straight Insertion Sort* sort ranges and one `std::stable_sort` range. This is also fast, though as soon as there is a $n \cdot \log n$ algorithm sort range envolved, it is no guarantee for being faster as a complete sort by the $n \cdot \log n$ algorithm. The GPU version shows that for CUDA compute compatibility 3.5 in figure 7.3.

In the funnel scene there are even up to three *Straight Insertion Sort* sort ranges, while there is no `std::stable_sort` sort range. In this case the algorithm is also fast (see figure 7.2) but a real advantage at this point is parallelism: The GPU version will execute those three sort ranges in parallel at the same time, so the whole execution time will be as long as its longest subset execution time.

Overall, a higher number of effective *Straight Insertion Sort* sort ranges though also means that the algorithm has to perform a higher number of merges. In the case of two or three sort ranges *AdaptiveFrameSort* will perform a heuristical merge only around the sort range borders, but if there are more than three sort ranges, this heuristical will not be performed and execution times get abruptly higher. At the same time, only one effective *Straight Insertion Sort* sort range can also mean that the entire frame is sorted by *Straight Insertion Sort*, which is very fast.

7.4 Conclusions Based on Measures of Unsortedness

Comparing the CPU results from figures 7.1, 7.2 and 7.3 with the scenes' analyses based on measures of unsortedness mentioned at the beginning of this work, it becomes very clear how the used algorithms work.

For the clothball *Timsort*'s timing curve looks very similar to the *rem* measure belonging to the scene (see figure 5.1). In fact, *Timsort* makes use of a *Natural Merge Sort* (see also section 2.4.8); Comparing this alone, a *Straight Merge Sort* is already optimal to the *runs* measure and adaptive to *inv* and *rem* (see also section 2.1.2.2). *Timsort*'s additional methods, like an *Exponential Search*, seem to make it mostly affected by the *rem* measure. `std::stable_sort`'s timing curve looks very similar to the scene's *runs* measure. Therefore the two algorithms make use of different measures of unsortedness. Looking at *AdaptiveFrameSort*, this newly introduced algorithm takes the advantage of both measures, whereas it does NOT take their disadvantages. This is very clearly visible for the funnel scene: Between frames 30 and 130, *AdaptiveFrameSort* has a similarly low runtime as *Timsort*, after frame 130 the *rem* unsortedness gets higher a lot (up to 76.3%) while the *runs* measure will not exceed 15.8% of unsortedness (see figure 5.2). In the same manner *AdaptiveFrameSort* does not follow *Timsort*'s runtime but — since it actually uses `std::stable_sort` as fallback — changes its measure of unsortedness to the less exhausting

runs measure. In case it uses *Straight Insertion Sort* (in addition to `std::stable_sort`), the measure is exactly that of *inv* (as noted before already). This measure does not get affected as vastly as any of the other measures. This means, it takes the advantage of both measures, *rem* and *runs*, but not their disadvantages.

7.5 Clothcar Details

The bigger scenes get, the more time a non-adaptive sorting algorithm will spend to sort their bounding boxes. In case of the clothcar scene with its 1,384,312 bounding boxes, Thrust's `thrust::stable_sort` takes around 9.5 ms with a Kepler graphics card to sort these. 9.5 ms do not leave much time for an algorithms overhead. Table 7.1 shows how *AdaptiveFrameSort*'s algorithm parts are distributed among the runtime. Note that the total times are sums of the part times. Furthermore, the part times are calculated from ONE run (based on the time-based adaptiveness of the algorithm it is hard to measure multiple runs, since the whole animation has to be restarted for that). If you want to have the exact execution times of *AdaptiveFrameSort*, rather refer to figure 7.1.

	CPU			CUDA CC 1.3		CUDA CC 3.5	
Time (ms)	AFS*	STL	Timsort	AFS*	Thrust	AFS*	Thrust
Memory Preparing	3.5			0.3		0.1	
Subset Sort (n-nsquare)	1.0			1.9		2.6	
Subset Sort ($n \cdot \log n$)	24.5			13.0		5.7	
Subset Merge	0.8			1.1		0.2	
Compute Sort Ranges	5.5			1.0		0.0	
Total	35.3	128.3	48.2	17.3	28.2	8.6	10.5

Table 7.1 Detailed *AdaptiveFrameSort* timings for the clothcar scene, frame 40. Number of bounding boxes: 1,384,312. Length of CPU/GPU subsets: n-nsquare: {484,505;415,302}, $n \cdot \log n$: 484,505. **AdaptiveFrameSort*

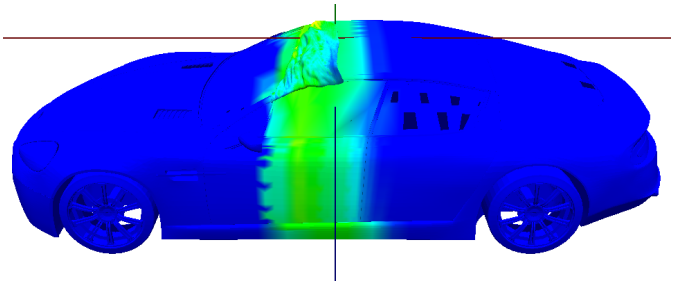

Figure 7.6 Clothcar scene, frame 40

It is clearly visible that the mere sort of the n-nsquare subsets takes very little time compared to the $n \cdot \log n$ subset sort. This is based on the fact that the with the car colliding cloth is very complex (345,984 bounding boxes). This is why the $n \cdot \log n$ subset consists of around 35% of all bounding boxes (the cloth's bounding boxes plus on the X-axis projected colliding car bounding boxes). It is nice to see that the GPU version of *AdaptiveFrameSort* uses parallelization very well in moments it can be used. The memory preparation takes only 0.3 ms, whereas the CPU version has to consecutively go through every element once, which results in 3.5 ms of time. In addition, the computation of the next frame's sort ranges takes 5.5 ms on CPU, while the CUDA version only takes 1 ms, which is very little compared to the 28.2 ms `thrust::stable_sort` takes to sort all bounding boxes (CUDA compute compatibility 1.3).

For CUDA compute compatiblity 3.5 we can measure a speed-up in many of the algorithm's parts. Memory preparing takes only 0.1 ms instead of 0.3 ms in compute compatibility 1.3. A real win can be found in the subset merge: 0.2 ms instead of 1.1 ms. The n-nsquare sort time remained roughly the same (2.6 ms instead of 1.9 ms). In fact, Thrust's sorting algorithm also gained a vast speed-up, which makes it harder to compete with it. While *AdaptiveFrameSort* had 28.2 ms - 13.0 ms = 15.2 ms of time for using its fast sides, now it only has 10.5 ms - 5.7 ms = 4.8 ms of time. Taking that the current *Cocktailsort* algorithm takes 2.6 ms for its subsets, there are only 2.2 ms left for the overall algorithm's overheard plus some extra time that we would like to become faster than the original Thrust algorithm. This can only mean one thing: The clothcar scene is still not big enough to have a real gain from *AdaptiveFrameSort*, respectively, the cloth covers too much space on the X-axis, which results in relatively too much time used for sorting the $n \cdot \log n$ subset.

Overall, for less cloth polygons or smaller cloth sizes (relatively to the car), *AdaptiveFrameSort* could be still faster. In summary, it is very helpful for *AdaptiveFrameSort* if it can sort edges very fast and the input data's middle range slowly with a non-adaptive algorithm. In this case (either 2 or 3 sort ranges exist) the merge heuristic will be effective and can merge the sorted subsets very quickly.

7.5.1 Average Timings For All Frames

The previous section investigated single steps of the algorithm. This section is interested in overall timings for the whole animation sequence. Figure 7.7 shows results for all examined algorithms, based on average execution times over all 150 frames after 100 runs. Each animation run (which consists of 150 frames for the clothcar scene) delivers a total execution time and the animation will be run 100 times in order to create an average execution time over all runs.

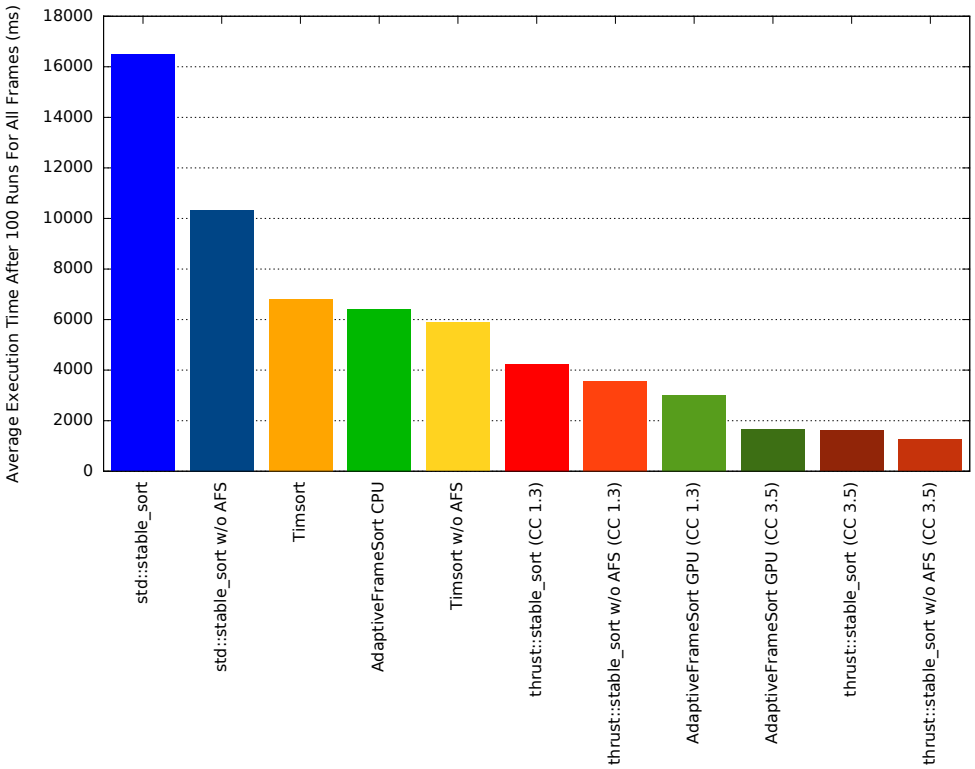

Figure 7.7 Average timings for the clothcar scene and any examined algorithm. The displayed timings are an average over all 150 frames being run 100 times. Sorted from slowest to fastest algorithm on the X-axis

On CPU, *AdaptiveFrameSort* is around 1.6 times faster than `std::stable_sort` sorting input data without additional overhead (marked as "w/o AFS", meaning without *AdaptiveFrameSort* data, which is an extra integer variable that resides in the data structure, see also section 7.2) and it is competitive to *Timsort*, although *Timsort* is slighly faster for sorting without overhead. Being 552 ms faster, *AdaptiveFrameSort* beats `thrust::stable_sort` (without additional overhead) on GPU with CUDA compute compatibility 1.3, making it an actual adaptive sorting algorithm on

GPU. Using CUDA compute compatibility 3.5 though, *AdaptiveFrameSort* cannot sort as fast as `thrust::stable_sort`; altogether, it is around 408 ms slower but still faster than all other tested sorting algorithms.

7.6 Funnel Details

7.6.1 Average Timings For All Frames

Just as shown in section 7.5.1, this section will investigate overall timings for the funnel scene, as well. Figure 7.8 shows the results.

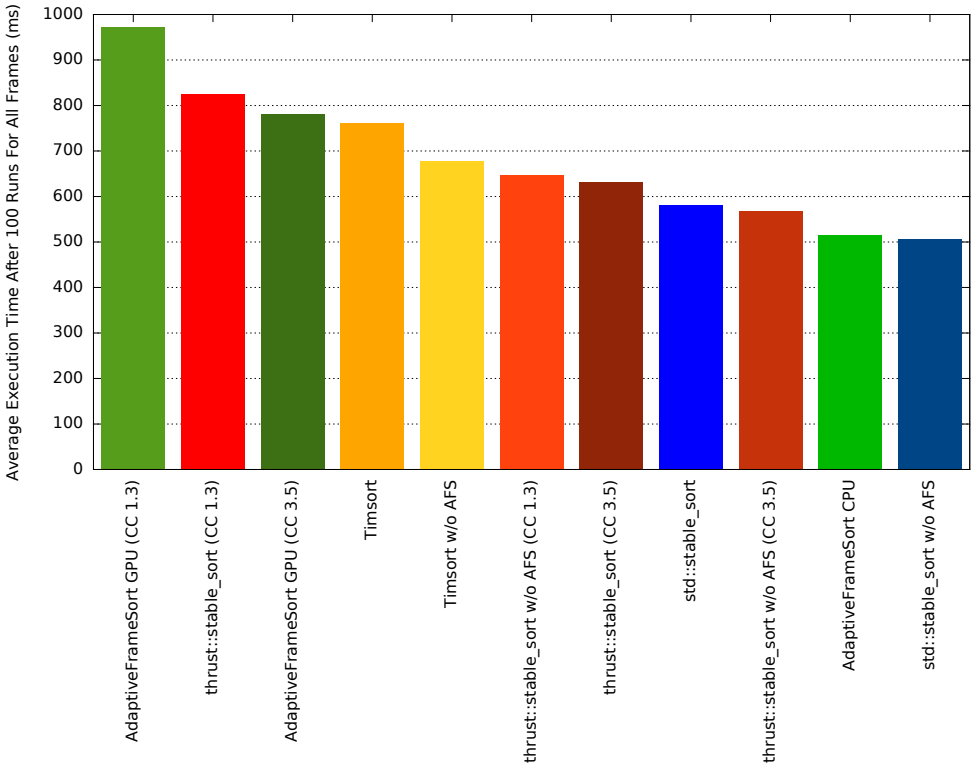

Figure 7.8 Average timings for the funnel scene and any examined algorithm. The displayed timings are an average over all 500 frames being run 100 times. Sorted from slowest to fastest algorithm on the X-axis

Whereas for the clothcar scene the slowest algorithm was `std::stable_sort` (see figure 7.7), it is

interesting to mention that in the funnel scene the same algorithm turns out to be the fastest sorter (sorting only the bounding boxes, without having additional *AdaptiveFrameSort* payload in the data structure). Compared to the winning overall time of 505 ms, *AdaptiveFrameSort* is around 8.5 ms slower but 66.4 ms faster than in the case that `std::stable_sort` sorts the same input data as *AdaptiveFrameSort*, which makes it competitive. Both, GPU algorithms (Thrust's as well as especially *AdaptiveFrameSort*) and the *Timsort* implementation lack of sorting speed in the funnel scene.

7.7 Clothball Details

Among other details, the previous sections investigated overall timing results for the scenes clothcar and funnel. As stated before in section 7.3 already, the clothball scene is not very handy for a faster sorting using an adaptive algorithm, though. For this reason, the section will only examine two representative frames closely in order to demonstrate *AdaptiveFrameSort*'s strengths and weaknesses rather than showing overall timing results.

Tables 7.2 and 7.3 refer to figure 6.8. Just according to the sort ranges displayed in the figure, the sorting of frame 4 (see table 7.2) is very fast for *AdaptiveFrameSort* (0.6 ms compared to STL's 5.1 ms). The adaptive *Timsort* is a comparably fast (also 0.6 ms). With 0.5 ms the computation of the next sort ranges is quite fast, though compared to the 0.6 ms overall sorting time it is too high.

Time (ms)	AFS CPU	STL	Timsort
Memory Preparing	0.1		
Subset Sort (n-nsquare)	0.0		
Subset Sort ($n \cdot \log n$)	0.0		
Subset Merge	0.0		
Compute Sort Ranges	0.5		
Total	0.6	5.1	0.6

Table 7.2 Detailed *AdaptiveFrameSort* timings for the clothball scene, frame 4. Number of bounding boxes: 184,460. Length of CPU subsets: n-nsquare: {92,230;92,230}

Frame 5 (see table 7.3) reveals the full functionality of *AdaptiveFrameSort*. Frame 4 suggested to use two equally sized n-nsquare sort ranges for frame 5. In frame 5 though, these were surprised

by two much movement at their edges (at the end of range 1 and at the beginning of range 2). This leads to an exceeding of the InvThreshold and a fallback to $n \cdot \log n$ sort. Since the algorithm merges both fallen back ranges, only one $n \cdot \log n$ sort remains. The execution now takes the same time as a regular $n \cdot \log n$ sort plus *AdaptiveFrameSort*'s overhead for the memory preparation, the initially launched n-nsquare algorithms and the computation of new sort ranges. Here, this overhead is 0.6 ms high and the algorithm's time will be slightly higher than the original STL algorithm.

Time (ms)	AFS CPU	STL	Timsort
Memory Preparing	0.1		
Subset Sort (n-nsquare)	0.0*		
Subset Sort ($n \cdot \log n$)	5.4		
Subset Merge	0.0		
Compute Sort Ranges	0.5		
Total	6.0	5.7	3.5

Table 7.3 Detailed *AdaptiveFrameSort* timings for the clothball scene, frame 5. Number of bounding boxes: 184,460. Length of CPU subsets: n-nsquare: {27,669;9,223;9,223; 9,223;64,561}, $n \cdot \log n$: {9,223;36,892;18,446}. *after fallback / exceeding the InvThreshold

Chapter 8

Conclusion

8.1 General Conclusion

A goal of this thesis was, to implement a novel time-based adaptive sorting algorithm. This goal succeeded.
In order to represent "real world data", two also in other papers used cloth animations were chosen, analyzed for pre-sorted data and their into 1D projected AABBs were consecutively sorted by different algorithms.

On CPU, the presented algorithm turned out to be faster than the adaptive hybrid algorithm *Timsort*, measured as a sum of sorting times over all frames in the representative animations. In a third animation type, a high-polygon cloth falls down on a high-polygon rigid and not moving car body. In this scene, *Timsort* was slightly faster. Both algorithms are hybrid and work in a similar way, which makes the comparison valid in a detailed level. *Timsort* uses a combination of *InsertionSort* and *Natural Merge Sort*, whereas *AdaptiveFrameSort* uses a combination of *Straight Insertion Sort*, STL's `std::stable_sort` (which uses a combination of *InsertionSort*, *Merge With Buffer* and *Adaptive In-Place Merge*) and STL's *Adaptive In-Place Merge*. *AdaptiveFrameSort*'s approach is rather a more generalizing; it utilizes existing algorithms "out-of-the-box", having its core features by exploiting fast algorithms and developing efficient strategies for using them.

AdaptiveFrameSort not only represents a hybrid algorithm, through its general approach it is easy to port to GPU streaming architectures. For the representative cloth animations, for older CUDA compute compatibilities (1.3, Tesla architecture), it shows similar results as the CPU version. For the additional clothcar scene, again, it shows similar results and after 150 frames already it gains a distinct win compared to Thrust's *Merge Sort*. For the more recent CUDA compute compatibility 3.5 (Kepler architecture), the algorithm offers rather poor results. In one of the representative scenes with a high rate of sub-algorithm switches, it results in a distinct loss compared to Thrust's non-adaptive algorithm. Also in the clothcar scene, it cannot achieve a win. Altogether, it means that the algorithm — as it is — succeeds in being adaptive on

GPU, which already puts a value to GPU sorting algorithms. Most established algorithms focus on highly optimized parallelism, rather than on adaptiveness. In terms of speed on GPU, *AdaptiveFrameSort* has to develop still for being competitive with recent architectures (which will be explained more closely in the next chapter).

At this point, though, it is important to mention that many algorithms presented in previous works were developed in an early state of CUDA: The hybrid GPU sort presented in [SA08] was published in the year 2008, using an NVIDIA GeForce 8600GTS graphics card, which deploys CUDA compute compatibility 1.1. In the paper, *Radix Sort* [SHG09] took about 600 ms for sorting $8 \cdot 10^6$ elements, whereas in a test run on hardware with CUDA compute compatibility 3.5 (an NVIDIA GeForce GTX 780) sorting the same amount of elements, Thrust's *Radix Sort* only takes 9 ms. The hybrid algorithm itself copied data from GPU to CPU memory (which also happens in a small extend for *AdaptiveFrameSort*); in the paper, it represented a small fraction of execution time. Nowadays, these memory copying times would account for a bigger part in execution times (for example, 9 ms do not leave a big gap for overhead). This means, that in more recent versions, CUDA offers a higher variety of instructions, as well as better caching techniques and overall faster underlying hardware. *AdaptiveFrameSort* and its utilized *Cocktailsort* produce too much overhead compared to Thrust's highly optimized *Merge Sort*.

Concluding, in its already optimized implementation, *AdaptiveFrameSort* paves the way for the introduction of time-based adaptive algorithms and adaptive GPU algorithms at the same time.

8.2 Future Work

The in this paper presented algorithm proved itself to be practically usable in many situations. Nevertheless, it offers much possibility for improvements. In the following course, some aspects will be listed.

Through the forecast of timings, *AdaptiveFrameSort* profits from hardware-specific parameters. In the course of the thesis, these parameters were established for a CPU system, a graphics card with CUDA compute compatibility 1.3 and a graphics card with CUDA compute compatibility 3.5. An idea for imrovement is, to introduce a method for setting these parameters based on an analysis of the underlying hardware in each concrete sorting case. This analysis only has to be run once on each new hardware component and should improve boundaries for the algorithm.

On CPU, the algorithm is implemented only in a serial way. First of all, this makes it more comparable to other serial sorting algorithms. In fact, just as for the GPU version, many parts of the algorithm could be parallelized on CPU as well: Range sums could be calculated in parallel, n-nsquare and nlogn sort ranges could be sorted fully in parallel, each sort range separately in one CPU thread. Although merging the sort ranges is already adaptive (especially with the local

merge heuristic) and STL's *In-Place Merge* is already in-place and adaptive, a global merge of all sort ranges could be run concurrently, maybe in a manner similar to *Parallel Prefix Sums* in [HSO07].

The same argument in a wider scale counts for the GPU version: Merging the sort ranges is currently even non-adaptive (from sides of Thrust's *Merge*); some preceding scans might generate a speed-up, considering that most sort ranges would not have much overlap to each other.

One of the most necessary improvements for this algorithm is the following: *Straight Insertion Sort* for the CPU already has a linear runtime compared to the *inv* measure. The GPU's *Cocktailsort* though, offers only super-linear time in this case. Its implementation only utilizes one CUDA block at a time and compared to Thrust's *Merge Sort*, for small input data, it has to much overhead. Whereas for realistic 270,000 elements on CPU *Straight Insertion Sort* only meets STL *Stable Sort*'s execution time for around five million inversions, for the same amount of elements on GPU, *Cocktailsort* meets Thrust *Merge Sort*'s execution time for already 500 (!) inversions. This implies, that in the representative animations, which involve a lot of relative movement, *AdaptiveFrameSort* cannot even be effective on GPU. In the past, major research did not focus on implementing efficient n-nsquare algorithms on GPU. In this thesis, a first step towards doing that has been done. As future work, a multi-block version with less overhead while utilizing more state-of-the-art CUDA instructions, would help improve the presented hybrid algorithm decisively on GPU.

Appendix A

Appendix

A.1 List of Figures

A.2 List of Tables

A.3 Bibliography

[AL90] Arne Andersson and Tony W Lai. "Fast updating of well-balanced trees". In: *SWAT 90*. Springer, 1990, pp. 111–121.

[Bar92] David Baraff. "Dynamic Simulation of Non-Penetrating Rigid Bodies". PhD thesis. Cornell University, 1992.

[Bat68] Kenneth E Batcher. "Sorting networks and their applications". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM. 1968, pp. 307–314.

[Ber97] Gino van den Bergen. "Efficient collision detection of complex deformable models using AABB trees". In: *Journal of Graphics Tools* 2.4 (1997), pp. 1–13.

[Big+08] Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg. "An Experimental Study of Sorting and Branch Prediction". In: *J. Exp. Algorithmics* 12 (June 2008), 1.8:1–1.8:39. ISSN: 1084-6654. DOI: *10.1145/1227161.1370599*. URL: *http://doi.acm.org/10.1145/1227161.1370599*.

[Ble14] Blendswap. *Aston Martin Rapide cycles*. 2014. URL: *http://www.blendswap.com/blends/view/26712*.

[BN89] G. Bilardi and A. Nicolau. "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines". In: *SIAM Journal on Computing* 18.2 (1989), pp. 216–228. DOI: *10.1137/0218014*.

[Cap+09] Gabriele Capannini, Fabrizio Silvestri, Ranieri Baraglia, and Franco Maria Nardini. "Sorting using bitonic network with CUDA". In: *Proc. LSDS-IR* (2009), pp. 33–40.

[CLP93] Svante Carlsson, Christos Levcopoulos, and Ola Petersson. "Sublinear merging and natural mergesort". English. In: *Algorithmica* 9.6 (1993), pp. 629–648. ISSN: 0178-4617. DOI: *10.1007/BF01190160*. URL: *http://dx.doi.org/10.1007/BF01190160*.

[CT08] Daniel Cederman and Philippas Tsigas. "A practical quicksort algorithm for graphics processors". In: *Algorithms-ESA 2008*. Springer, 2008, pp. 246–258.

[Dij82] Edsger W. Dijkstra. "Smoothsort, an alternative for sorting in situ". In: *Science of Computer Programming* 1.3 (1982), pp. 223–233. ISSN: 0167-6423. DOI: *http://dx.doi.org/10.1016/0167-6423(82)90016-8*.

[Dom11] Ken Domino. *Lecture 4: Introduction to Parallel Computing Using CUDA*. IEEE Boston Continuing Education Program, 2011. URL: *http://domemtech.com/ieee_pp/Lecture4.pdf*.

[EF03] Amr Elmasry and Michael L Fredman. "Adaptive sorting and the information theoretic lower bound". In: *STACS 2003*. Springer, 2003, pp. 654–662.

[Elm02] Amr Elmasry. "Priority queues, pairing, and adaptive sorting". In: *Automata, Languages and Programming*. Springer, 2002, pp. 183–194.

[Elm04] Amr Elmasry. "Adaptive sorting with AVL trees". In: *Exploring New Frontiers of Theoretical Informatics*. Springer, 2004, pp. 307–316.

[Eri05] Christer Ericson. *Real-time collision detection*. Vol. 14. Elsevier Amsterdam/Boston, 2005.

[EW92] Vladmir Estivill-Castro and Derick Wood. "A Survey of Adaptive Sorting Algorithms". In: *ACM Comput. Surv.* 24.4 (Dec. 1992), pp. 441–476. DOI: *10.1145/146370.146381*. URL: *http://doi.acm.org/10.1145/146370.146381*.

[Fou14] Python Software Foundation. *Python Sorting*. 2014. URL: *https://docs.python.org/2/howto/sorting.html*.

[Fuj12] Goro Fuji. *cpp-TimSort*. 2012. URL: *https://github.com/gfx/cpp-TimSort/blob/master/timsort.hpp*.

[GGK06] Alexander Greß, Michael Guthe, and Reinhard Klein. "GPU-based Collision Detection for Deformable Parameterized Surfaces". In: *Computer Graphics Forum*. Vol. 25. 3. Wiley Online Library. 2006, pp. 497–506.

[GLM96] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. "OBBTree: A hierarchical structure for rapid interference detection". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 171–180.

[Gov+05a] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. *GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management*. Tech. rep. MSR-TR-2005-183. Original November 2005, Revised March 2006. Microsoft Research, Dec. 2005, p. 14. URL: *http://research.microsoft.com/apps/pubs/default.aspx?id=64572*.

[Gov+05b] Naga K Govindaraju, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C Lin, and Dinesh Manocha. "Interactive collision detection

between deformable models using chromatic decomposition". In: *ACM Transactions on Graphics (TOG)*. Vol. 24. 3. ACM. 2005, pp. 991–999.

[Gui+77] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. "A New Representation for Linear Lists". In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. ACM, 1977, pp. 49–60. DOI: *10.1145/800105.803395*. URL: *http://doi.acm.org/10.1145/800105.803395*.

[GZ06] A. Greß and G. Zachmann. "GPU-ABiSort: optimal parallel sorting on stream architectures". In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. Apr. 2006, 10 pp. DOI: *10.1109/IPDPS.2006.1639284*.

[He+07] Bingsheng He, Naga K Govindaraju, Qiong Luo, and Burton Smith. "Efficient gather and scatter operations on graphics processors". In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM. 2007, p. 46.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel prefix sum (scan) with CUDA". In: *GPU gems* 3.39 (2007), pp. 851–876.

[Kap+00] Ujval J Kapasi, William J Dally, Scott Rixner, Peter R Mattson, John D Owens, and Brucek Khailany. "Efficient conditional operations for data-parallel architectures". In: *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM. 2000, pp. 159–170.

[Khr14] Khronos. *OpenCL*. 2014. URL: *https://www.khronos.org/opencl/*.

[Kim+09] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-eui Yoon. "HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs". In: *Computer Graphics Forum*. Vol. 28. 7. Wiley Online Library. 2009, pp. 1791–1800.

[Knu75] D.E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1975.

[KW05] Peter Kipfer and Rüdiger Westermann. "Improved GPU sorting". In: *GPU gems* 2 (2005), pp. 733–746.

[Lan12] Hans Werner Lang. *Algorithmen in Java: Sortieren, Textsuche, Codierung, Kryptografie*. 3. Aufl. Informatik 10-2012. Online-Ressource (XV, 393 S.) München: Oldenbourg, 2012. ISBN: 9783486718973. URL: *http://dx.doi.org/10.1524/9783486718973*.

[Le 07] S. Le Grand. "Broad-phase collision detection with CUDA". In: *GPU Gems 3*. Ed. by H. Nguyen. Addison-Wesley Professional, 2007.

[LMM10] Christian Lauterbach, Qi Mo, and Dinesh Manocha. "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries". In: *Computer Graphics Forum*. Vol. 29. 2. Wiley Online Library. 2010, pp. 419–428.

[LOS10] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. "GPU sample sort". In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–10.

[LP91] Christos Levcopoulos and Ola Petersson. "Splitsort—an adaptive sorting algorithm". In: *Information Processing Letters* 39.4 (1991), pp. 205–211.

[LP93] Christos Levcopoulos and Ola Petersson. "Adaptive heapsort". In: *Journal of Algorithms* 14.3 (1993), pp. 395–413.

[Man85] H. Mannila. "Measures of Presortedness and Optimal Sorting Algorithms". In: *Computers, IEEE Transactions on* C-34.4 (Apr. 1985), pp. 318–325. ISSN: 0018-9340. DOI: *10.1109/TC.1985.5009382*.

[McI93] Peter McIlroy. "Optimistic Sorting and Information Theoretic Complexity". In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '93. Society for Industrial and Applied Mathematics, 1993, pp. 467–474. ISBN: 0-89871-313-7. URL: *http://dl.acm.org/citation.cfm?id=313559.313859*.

[Meh88] K Mehlhorn. *Sortieren und Suchen, Band 1 von Datenstrukturen und effiziente Algorithmen*. 1988.

[MEP96] Alistair Moffat, Gary Eddy, and Ola Petersson. "Splaysort: Fast, versatile, practical". In: *Software: Practice and Experience* 26.7 (1996), pp. 781–797.

[MZ14] David Mainzer and Gabriel Zachmann. *Collision Detection Based on Fuzzy Scene Subdivision*. GPU Computing and Applications. Springer, 2014.

[NVI14a] NVIDIA. *CUDA*. 2014. URL: *https://developer.nvidia.com/cuda-zone*.

[NVI14b] NVIDIA. *CUDA Features and Technical Specifications*. 2014. URL: *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications*.

[OW12] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen, 5. Auflage*. Spektrum Akademischer Verlag, 2012. ISBN: 978-3-8274-2803-5. DOI: *10.1007/978-3-8274-2804-2*. URL: *http://dx.doi.org/10.1007/978-3-8274-2804-2*.

[Pet02a] Tim Peters. *Timsort*. 2002. URL: *http://bugs.python.org/file4451/timsort.txt*.

[Pet02b] Tim Peters. *Timsort*. 2002. URL: *https://mail.python.org/pipermail/python-dev/2002-July/026837.html*.

[PM92] Ola Petersson and Alistair Moffat. "A framework for adaptive sorting". English. In: *Algorithm Theory — SWAT '92*. Ed. by Otto Nurmi and Esko Ukkonen. Vol. 621. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 422–433. ISBN: 978-3-540-55706-7. DOI: *10.1007/3-540-55706-7_38*. URL: *http://dx.doi.org/10.1007/3-540-55706-7_38*.

[PSL10] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. "Fast in-place sorting with cuda based on bitonic sort". In: *Parallel Processing and Applied Mathematics*. Springer, 2010, pp. 403–410.

[PSL12] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. "A novel sorting algorithm for many-core architectures based on adaptive bitonic sort". In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE. 2012, pp. 227–237.

[Pur+03] Timothy J Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. "Photon mapping on programmable graphics hardware". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association. 2003, pp. 41–50.

[SA08] Erik Sintorn and Ulf Assarsson. "Fast parallel GPU-sorting using a hybrid algorithm". In: *Journal of Parallel and Distributed Computing* 68.10 (2008), pp. 1381–1388.

[Sch61] C. Schensted. "Longest increasing and decreasing subsequence". In: *Canadian Journal of Mathematics* 13 (1961), pp. 179–191. DOI: *10.4153/CJM-1961-015-3*.

[Sch80] Jacob T Schwartz. "Ultracomputers". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.4 (1980), pp. 484–521.

[Sed98] Robert Sedgewick. *Algorithms in c++, parts 1-4 (fundamental algorithms, data structures, sorting, searching)*. Addison-Wesley, 1998.

[Sen+07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. "Scan primitives for GPU computing". In: *Graphics Hardware*. Vol. 2007. 2007, pp. 97–106.

[SGI14a] SGI. *inplace_merge*. 2014. URL: *https://www.sgi.com/tech/stl/inplace_merge.html*.

[SGI14b] SGI. *sort*. 2014. URL: *https://www.sgi.com/tech/stl/sort.html*.

[SGI14c] SGI. *stable_sort*. 2014. URL: *https://www.sgi.com/tech/stl/stable_sort.html*.

[SGI14d] SGI. *STL*. 2014. URL: *https://www.sgi.com/tech/stl/download.html*.

[SHG09] N. Satish, M. Harris, and M. Garland. "Designing efficient sorting algorithms for manycore GPUs". In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. May 2009, pp. 1–10. DOI: *10.1109/IPDPS.2009.5161005*.

[Ski88] StevenS. Skiena. "Encroaching lists as a measure of presortedness". English. In: *BIT Numerical Mathematics* 28.4 (1988), pp. 775–784. ISSN: 0006-3835. DOI: *10.1007/BF01954897*. URL: *http://dx.doi.org/10.1007/BF01954897*.

[SS93] R. Schaffer and R. Sedgewick. "The Analysis of Heapsort". In: *Journal of Algorithms* 15.1 (1993), pp. 76–100. ISSN: 0196-6774. DOI: *http://dx.doi.org/10.1006/jagm.1993.1031*.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[Tan+11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. "Collision-streams: fast gpu-based collision detection for deformable models". In: *Symposium on interactive 3D graphics and games*. ACM. 2011, pp. 63–70.

[UNC14] UNC. *UNC Dynamic Scene Benchmarks*. 2014. URL: *http://gamma.cs.unc.edu/DYNAMICB/*.

[Van91] Allen Van Gelder. *Simple adaptive merge sort*. 1991.

[WZ09] Rene Weller and Gabriel Zachmann. "Inner sphere trees for proximity and penetration queries." In: *Robotics: Science and Systems*. Vol. 2. 2009.

[Ye+10] Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and Paolo Ienne. "High performance comparison-based sorting algorithm on many-core GPUs". In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–10.

[Ye+11] Yin Ye, Zhihui Du, David A Bader, Quan Yang, and Weiwei Huo. "GPUMemSort: A High Performance Graphic Co-processors Sorting Algorithm for Large Scale In-Memory Data". In: *GSTF International Journal on Computing* 1.2 (2011), pp. 23–28.