Exploiting Space and Time Coherence in Grid-based Sorting

Rubens Carlos Silva Oliveira[†], Claudio Esperança[†], Antonio Oliveira[†]
[†]COPPE - Federal University of Rio de Janeiro
Email: rubcarso@ibest.com.br, esperanc@cos.ufrj.br, oliveira@cos.ufrj.br

Abstract—**In recent years, many approaches for real-time simulation of physical phenomena using particles have been proposed. Many of these use 3D grids for representing spatial distributions and employ a collision detection technique where particles must be sorted with respect to the cells they occupy. In this paper we propose several techniques that make it possible to explore spatio-temporal coherence in order to reduce the work needed to produce a correct ordering and thus accelerate the collision detection phase of the simulation. Sequential and GPU-based implementations are discussed, and experimental results are presented. Although devised with particle-based simulations in mind, the proposed techniques have a broader scope, requiring only some means of establishing subsequences of the input which did not change from one frame to the next.**

Keywords-**GPU programming; sorting; particle based physics**

I. Introduction

The process of computing the interference between multiple objects moving through space is crucial in many applications, especially in simulations, physically based or not. The problem has worst case complexity $O(n^2)$, where n is the number of objects, but can be even harder for objects with non-trivial geometries. An approach that has been gaining popularity in the last few years is to approximate objects by collections of particles, typically represented by small spheres, leading thus to a problem of detecting the interference between all pairs of particles. The main advantages of this idea are that sphere-sphere intersection tests are cheap, and spatial decomposition schemes can be effectively applied to reduce the search to small spatial neighborhoods.

In this context, when considering an implementation on parallel architectures such as GPUs, the choice of regular 3D grids for organizing space is more appealing than hierarchical data structures. This is corroborated by several physical simulation schemes reported recently, e.g. [1], [2], [3], [4]. The overall idea of such schemes is that the neighborhood of a particle can be inferred by looking at the grid cells it occupies, and thus, interfering particles may be found by inspecting those cells or other neighboring cells. Thus, the approach relies on being able to perform two simple operations, namely, (1) given a particle, computing the cells it intersects, and (2) given a cell address, computing all particles intersecting it. Early schemes allowed for only a few particles being assigned to any one given cell [1] so as to perform operation (2) efficiently. This restriction was lifted from more modern schemes by simply sorting particles by their cell ids, so that all particles occupying a given cell are contiguous in the sorted array.

In this scenario, the sorting algorithm assumes a crucial importance, especially if we consider that, in many typical simulations, we may expect relatively few changes in the mapping between particles and the grid cells they occupy. In other words, since the input changes little from frame to frame, it seems logical to assume that algorithms that are able to explore time coherence will fare better than those which oblivious to this fact. It is important to note that, unlike in other applications, in this case it is possible to tell exactly what parts of the input have changed from one frame to the next, which makes it possible to pinpoint what parts must be re-sorted. This, in fact, is the only restriction for using the techniques discussed in this paper, which we feel have a broader scope than the particular application focused here.

In this paper, we propose several ideas that can be combined with the traditional implementations found in CPU- and GPU-based libraries so as to obtain significant performance enhancement in particle-based simulations. These ideas are explained in detail and the results of several experiments conducted in different system architectures are presented.

II. Related work

Collision detection is a heavily studied topic in Computer Graphics, with proposed solutions that vary widely depending on issues like (1) the types of objects being considered, e.g., rigid, deformable, fluids, etc, (2) whether discrete or continuous-time detection is desired, and (3) the types of computer architectures available, e.g., CPU-only, GPUs or other parallel architectures. The reader is referred to one of the many surveys of the field, such as [5], [6], [7].

In this paper, we are mostly interested in a particular technique using spatial grids. In [8], a 3D grid mapped on a texture is used to record up to four particle ids in a fully GPU-based simulation of a large number of rigid bodies. In [9], this idea is extended with the use of a spatial hash code that enables assigning more than one particle to any given spatial grid cell. Particles belonging to the same cell are then clustered together in an array by using sorting them by hash code. This technique is explained in more detail in SectionIII.

Sorting algorithms have also been extensively researched since the early days of Computer Science. Perhaps the most commonly used sorting algorithm present in general-purpose libraries is the *Quicksort* [10], usually implemented with modifications proposed by Sedgewick [11]. Although in the worst case it may make $O(n^2)$ comparisons, this is rare, and in practice it tends to perform faster than other $O(n \log n)$ algorithms due to the localized way it accesses memory. Also noteworthy are the facts that the Quicksort algorithm is in-place and non-stable.

Since we are interested in the problem for nearly sorted inputs, it is worth mentioning that Quicksort does not fare specially well in that case. The extremely informative page by David R. Martin [12] deems the Insertion Sort, an $O(n^2)$ algorithm, as the clear winner among the best known algorithms. Among the algorithms with $O(n \log n)$ time complexity, two adaptive approaches are worth mentioning. The first is *Timsort*, an algorithm devised in 2002 by Tim Peters [13] for use in the Python programming language. It finds sorted subsets of the input data and uses this information to sort the remainder more efficiently. The second is the algorithm known as *Smoothsort* [14], proposed as a variation of the *Heapsort*, is known to work well with nearly sorted inputs. It is a fairly complex algorithm which degrades smoothly (hence the name) from $O(n)$ in the best case, when the input is already sorted, to $O(n \log n)$ in the worst case. A more thorough analysis of adaptive sorting algorithms can also be found in [15].

When considering parallel architectures in general, and GPU-based programming libraries in particular, two other algorithms are more commonly used, namely the *Bitonic Sort* and the *Radix Sort*. In our tests, we used implementations of these algorithms included in NVidia's SDK [16].

The Bitonic Sort [17] consists of building short bitonic subsequences (up to 4 elements initially) and then progressively merging them together. A bitonic sequence is a concatenation of two subsequences, one monotonically increasing and another monotonically decreasing. It runs in $O(n \log^2 n)$ time, but is very well-suited to parallel architectures since the pattern of comparisons does not depend on the way input data is layed out in memory.

Radix Sort is a sorting strategy which does not use comparison, but rather, groups keys based on the values of the individual digits in the same position. The idea is credited to Herman Hollerith who proposed its use in tabulating machines and to Harold H. Seward, who first proposed it formally as a computer algorithm in 1954. The algorithm is not restricted to integer keys, but can be used for any data that can be encoded positionally. Two main variants are recognized, the MSD (Most Significant Digit) variant examines digits in lexicographic order, being most suitable for sorting character strings. Conversely, the LSD (Least Significand Digit) variant examines digits from right to left, being suited for sorting integer numbers. In the experiments described in this paper, the Radix Sort follows ideas outlined in [18], an LSD variant which uses 4 bits digits.

III. Collision detection using spatial grids

This Section describes succinctly the scheme for particle collision detection proposed in [9] which will be used in the experiments for empirical validation of the techniques introduced in Section IV.

The scheme takes place in a world space delimited by an axis-aligned parallelepiped divided into a tridimensional grid of cubical cells. The size of the grid is selected in such a way as to guarantee that any given particle (a sphere, rather) does not intersect more than eight cells of a $2 \times 2 \times 2$ neighborhood, i.e., the side of each cell is not smaller than twice the radius of the biggest particle. Let n_x, n_y and n_z be the number of cells in each dimension. Then, a given particle P is assigned to the cell which contains its center, say, cell with grid coordinates (i_x, i_y, i_z), which can be mapped into a single integer key in range $[0, n_x n_y n_z - 1]$ through a hash function such as

$$hash(i_x, i_y, i_z) = i_x + n_x(i_y + n_y i_z).$$

Once particles are assigned to cells, the array of particles is then sorted with respect to their hashed keys, so that particles assigned to the same cell are contiguous. The sorted array is then scanned to determine the ranges of positions that hold particles with the same key. Let $F(h)$ and $L(h)$ be the first and last indices of the sorted array that hold particles hashed to grid position h, then these two values are stored in an array at position h so that they can be accessed in constant time. The process of building the required data structures is illustrated in Algorithm 1 Thus, a particle assigned to a given cell must

Algorithm 1 Build data structures for grid-based collision detection

```
Input: $P$ array with the positions of the $n$ particles
Input: $n_x, n_y, n_z$ the dimensions of the grid
  /* Build $H$ */
  $H \leftarrow$ an array with $n$ positions
  for $i = 0 \rightarrow n - 1$ do
    $i_x, i_y, i_z \leftarrow$ cell coordinates of $P[i]$
    $h \leftarrow hash(i_x, i_y, i_z)$
    $H[i] \leftarrow (h, i)$
  end for
  SORT $H$ so that $H[i].h \leq H[i+1].h, \forall i$
  /* Build $F,L$ */
  $F, L \leftarrow$ arrays with $n_x n_y n_z$ positions all set to $-1$
  $h \leftarrow Pair[0].h$
  $F(h), L(h) \leftarrow 0, 0$
  for $i = 1 \rightarrow n - 1$ do
    if $h = H[i].h$ then
      $L[h] \leftarrow i$
    else
      $F(h), L(h) \leftarrow i, i$
      $h \leftarrow H[i].h$
    end if
  end for
```

be tested for intersection with particles assigned either to the same cell or to one of the immediate neighbor cells, i.e, in a $3 \times 3 \times 3 = 27$-cell neighborhood.

IV. Optimized particle sorting in CPU

Two main strategies are discussed in this section with respect to the optimization of the sort phase of grid-based schemes. The first one consists of using adaptive sort algorithms, i.e., algorithms which perform better with nearly sorted collections. The second strategy consists of pre-conditioning the input of non-adaptive sort algorithms by splitting it into ordered and non-ordered subsequences, sorting the latter and merging it into the former.

A. *Using adaptive sorters*

An examination of Algorithm 1 reveals that the input collection submitted to the sort algorithm is manufactured from scratch at each iteration. Clearly, then, the first required modification is to apply the sort algorithm to an array which does not change overmuch from one frame to the next. This can be achieved by using another level of indirection when referencing the H array. The idea is to define another array, say I, which will contain indices to array H. At the beginning of the simulation I is initialized so that $I[i] = i$. At each frame, array I is sorted so that $H[I[i]].h \le H[I[i+1]].h$, for $0 \le i < n-1$. In fact, since H is never reordered, there is no need to use the $.i$ field at all, so that H merely stores hash values. Once I is sorted at the end of one frame it can be used again as a good initial guess for sorting H for the next frame. Algorithm 2 reflects the proposed changes.

Algorithm 2 Revised version of Algorithm 1 using nearly sorted arrays

```
Input: $P$ array with the positions of the $n$ particles
Input: $I$ array with a permutation of $\{0..n-1\}$ from previous frame
Input: $n_x, n_y, n_z$ the dimensions of the grid
  /* Build $H$ */
  $H \leftarrow$ an array with $n$ positions
  for $i = 0 \rightarrow n-1$ do
    $i_x, i_y, i_z \leftarrow$ cell coordinates of $P[i]$
    $H[i] \leftarrow hash(i_x, i_y, i_z)$
  end for
  SORT $I$ so that $H[I[i]] \le H[I[i+1]], \forall i$
  /* Build $F$,$L$ */
  $F, L \leftarrow$ arrays with $n_x n_y n_z$ positions all set to $-1$
  $h \leftarrow H[I[0]]$
  $F(h), L(h) \leftarrow 0, 0$
  for $i = 1 \rightarrow n-1$ do
    if $h = H[I[i]]$ then
      $L[h] \leftarrow i$
    else
      $F(h), L(h) \leftarrow i, i$
      $h \leftarrow H[I[i]]$
    end if
  end for
```

B. *Split and Merge*

Given that the best-performing sort algorithms are not adaptive, another strategy for taking advantage of nearly sorted inputs is to split it into two collections: a sequence which is known to be already sorted and another, hopefully much smaller, which must be sorted from scratch. Unlike in other applications, the use of a grid provides a simple way to obtain a sorted subset of the input, namely, by selecting all particles which have not crossed a cell boundary from one frame to the next. Thus, we propose a modification of Algorithm 2, where array I is first split into two subsequences, both stored into an auxiliary array S: the sorted subsequence is placed at the beginning of the array, while the indices of particles which moved to another cell are placed at the end of the array. The unordered subsequence is then sorted and, finally, both sorted subsequences can now be merged back into array I. This idea is shown in a more formal way in Algorithm 3.

Algorithm 3 Modification of Algorithm 2 using a Split-Merge strategy

```
Input: $P$ array with the positions of the $n$ particles
Input: $I$ array with a permutation of $\{0..n-1\}$ from previous frame
Input: $H$ array with hash values from the previous frame
Input: $n_x, n_y, n_z$ the dimensions of the grid
  /* Rebuild $H$ and $I$ – Split Phase */
  $S \leftarrow$ array with $n$ positions
  $m \leftarrow 0$
  for $j = 0 \rightarrow n-1$ do
    $i \leftarrow I[j]$
    $i_x, i_y, i_z \leftarrow$ cell coordinates of $P[i]$
    $h \leftarrow hash(i_x, i_y, i_z)$
    if $h = H[i]$ then
      /* Particle in the same cell */
      $S[m] \leftarrow i$
      $m \leftarrow m+1$
    else
      /* Particle moved: store at the end of $S$ */
      $S[n-1-j+m] \leftarrow i$
      $H[i] \leftarrow h$
    end if
  end for
  /* Merge Phase */
  SORT $S[i]$ for $i \in \{m..n-1\}$ so that $H[S[i]] \le H[S[i+1]]$
  MERGE $S[0..m-1]$ and $S[m..n-1]$ into $I$
  /* Build $F$,$L$ */
  ... identical to Algorithm 2 ...
```

V. GPU Split and Merge

The Split and Merge strategy as outlined in Algorithm 3 provides a way of leveraging the performance of the two most common GPU-based sort implementations, namely, the Bitonic and Radix sort algorithms, onto nearly sorted inputs. This, however, depends on devising parallel implementations of the split and merge operations. Fortunately, both operations have been the subject of intense research since the popularization of GPUs. In particular, the seminal work of Blelloch [19] describes several parallel primitive operations which can be used to implement a wide variety of algorithms. In fact, some

$I =$	5	7	3	1	4	2	7	2
Changed =	F	F	F	F	T	T	F	T
1^{st} scan =	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	4	4	$\boxed{4}$	5
2^{nd} scan =	4	4	4	4	$\boxed{5}$	$\boxed{6}$	6	$\boxed{7}$
$S =$	5	7	3	1	7	4	2	2

Fig. 1. Example of the parallel split operation of I into S (adapted from [20].

libraries already contain implementations of parallel primitives such as *scan* and *prefix scans*.

For the experiments in this paper we have developed two GPU implementations of the Split and Merge strategy, which are discussed below. Both use the same implementation for the Split operation, but differ on the Merge phase.

A. GPU implementation 1

The Split operation was implemented according to the approach proposed by Blelloch [19], which in turn is the direct application of two *scan* operations plus one parallel permutation operation. In a nutshell, in the first scan, the algorithm computes an array containing an enumeration from 0 to $m-1$ for the unchanged elements. An enumeration is an array with the same size as the input array which is filled in parallel with non-decreasing values according to some rule. In our case, the rule is equivalent to setting a variable to 0, scanning the array I from left to right and incrementing the variable each time an unchanged element is found. The second scan produces an enumeration from m to $n-1$ using a back scan of the changed elements. This is equivalent to the setting a variable to $n-1$ and scanning I from right to left, decrementing the variable each time a changed element is found. Once this is accomplished, array S can be filled with the sought permutation of I by using either the first or the second enumeration as indices for the destination array. Figure 1 illustrates this procedure applied on an example.

The Merge operation was implemented according to the approach described in set of lecture notes by Ottmann [21], which is very similar to the parallel merge algorithm described later by Satish et al. [18]. In essence, given two sorted subsequences A and B, the algorithm consists of computing the rank of each element in the sorted sequence C. For an element $a_i \in A$, its rank in C, written $\text{rank}(a_i, C)$, is given by $i + \text{rank}(a_i, B)$, where $\text{rank}(a_i, B)$ is the number of elements in B smaller than a_i. This operation may be computed by performing a binary search for each element a_i in parallel. Clearly, the same idea applies to finding the rank of elements of B in A, i.e., $\text{rank}(b_i, A)$. If the size of each subsequence is relatively small, this approach is enough to obtain the desired merged sequence. For typical inputs, however, this does not scale well. The solution then is to split the problem into several small independent subproblems in the following way:

1) Split B into subsequences of no more than m elements, say.
2) For each subsequence B_i of B, find (in parallel) the rank of its last element $B_{i,m}$ in A and call it $\text{rank}(B_{i,m}, A)$.
3) Define a subsequence A_i as the elements of A at indices ranging from $\text{rank}(B_{i-1,m}, A)$ to $\text{rank}(B_{i,m}, A)$.
4) The subproblems which consist of merging A_i and B_i are independent and may be performed in parallel.

B. GPU implementation 2

The second GPU implementation uses a somewhat different way of obtaining the final sorted sequence. In particular, after the splitting phase, the subsequence corresponding to the particles moved to another cell is *not* sorted before merging. Rather, the sorting occurs *after* the merge. Another salient feature of this implementation is the use of atomic operations, i.e., instructions that ensure exclusive access to a resource, such as memory. Thus, concurrent accesses to the resource are serialized with no pre-established priority.

The approachcan be summarized as follows.

1) After the Split phase, let A refer to the already sorted portion of the split array and B refer to the unsorted portion. In the terminology used in Algorithm 3, these would be $S[0..m-1]$ and $S[m..n-1]$, respectively. For convenience, we assume that B is an array with $k = n - m$ elements.
2) Define an array C with $m+1$ positions. An element $C[i]$ will contain the number of elements of B that will be inserted in the sorted array between elements $A[i-1]$ and $A[i]$. Notice, in particular, that $C[0]$ will contain the number of elements of B that are smaller than $A[0]$, and $C[m]$ will contain the number of elements of B greater than $A[m-1]$.
3) Define an array R with k positions. The idea is to set $R[j]$ to the number of elements of A which are smaller than $B[j]$, i.e., at the end, $R[j] = \text{rank}(B[j], A)$ for $0 \le j < k$. Thus, in the sorted array, $B[j]$ should appear before $A[R[j]]$ and after $A[R[j]-1]$.
4) Define an array D with k positions. Each element $D[j]$ will contain the order of insertion of $B[j]$ among all elements of B to be inserted between $A[R[j]-1]$ and $A[R[j]]$.
5) In order to compute C, B and R, elements of B are examined in parallel. For a given $B[j]$, $R[j]$ is computed using binary search, and arrays C and D are updated using atomic operations. Figure 2(a) illustrates this portion of the algorithm for a sample input.
6) The final positions of each element of A are stored in an array E using a parallel sum scan of C. In other words, $A[i]$ is to be copied to the sorted array S at position $E[i] = i + \sum_{j=0}^{i-1} C[j]$.
7) Move elements of A and B to the result array S. Notice that, at this point, elements of A are indeed at their correct positions in S, while elements of B may not be. Figure 2(b) shows the result array after this step of the algorithm.
8) In order to obtain the final result, all elements of B which in S fell between two consecutive elements of A must be sorted. In other words, ranges of elements between $S[E[i]]$ and $S[E[i+1]]$ must be sorted. This

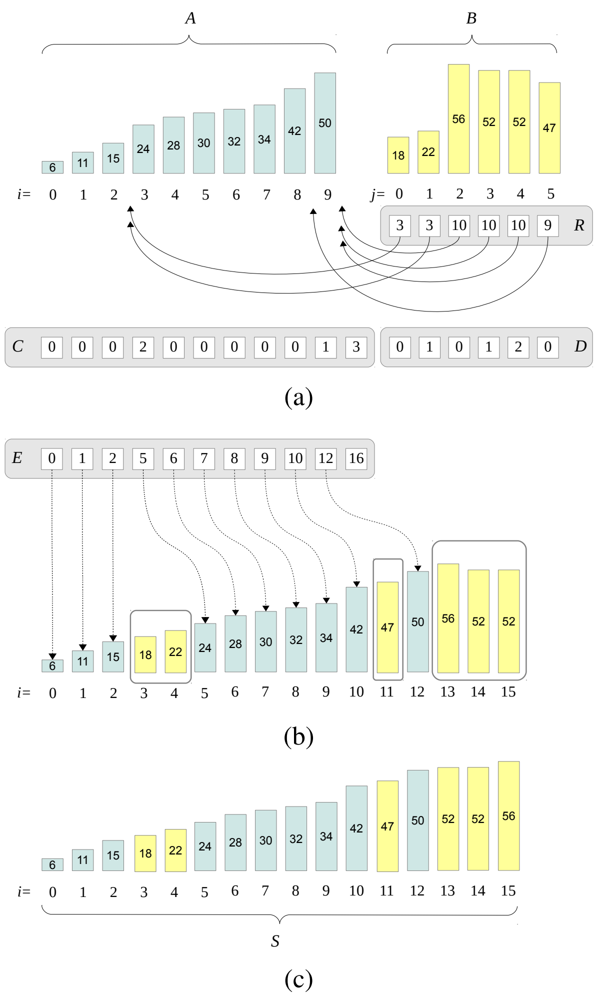

Fig. 2. An example run of the Parallel Merge described in Section V-B: (a) Original input and arrays R, C and D. (b) After elements of A and B have been copied to the output. (c) After subranges of the output containing elements of B have been sorted.

can be accomplished using a simple sequential sorting algorithm, given that such ranges must have relatively few elements. In our implementation we used Insertion Sort for this task. The final array S is shown in Figure 2(c).

The rationale that guided the development of this algorithm is that each subset of elements of B inserted between any two consecutive elements of A should be small if B, as a whole is small. Moreover, the problems of sorting these subsets are independent, and thus can be sorted by a separate thread in parallel. It stands to reason, then, that the overall time spent for sorting all subsets, being proportional to the size of the largest subset, should be relatively small even if an inefficient algorithm is used in each thread.

VI. Experiments and Results

Several experiments were conducted to try to assess the usefulness of the described techniques either by themselves or in the context of particle simulation applications. In the former case, tests consisted of measuring the relative speeds of the algorithms as the amount of "sortedness" of the input change. In the latter case, tests try to measure the influence of the various algorithms in two particle simulation applications developed for this purpose.

	System **A**	System **B**
System type	Desktop	Laptop
CPU	AMD Phenom II X4 940	Intel i7-2630QM
CPU Clock	3000MHz	1995MHz
CPU cores	4	8
Main memory	4GB	8GB
GPU	ATI Radeon 5870	NVidia GeForce GT540M

TABLE I
Technical characteristics of the systems used in the experiments.

All implementations were coded by the authors using C++ and OpenCL. Two computer systems were used for all tests, with technical characteristics as shown in Table I.

A. *Stand alone speedup tests*

The first batch of tests tried to establish how well traditional sorting algorithms fare with respect to each other and also with respect to the Split-Merge approach described in Section IV-B. All implementations are sequential (CPU only). The tests were conducted on System A for a collection of 256 thousand integer numbers. Figure 3 shows the results of sorting inputs with varying degrees of sortedness. For instance, an input with 10% modified values was obtained by scrambling 10% of an otherwise sorted array.

An inspection of Figure 3 suggests that among the traditional algorithms, QuickSort fares best, although it is not considered an adaptive algorithm, i.e., its asymptotic time complexity is not affected by how well the input is sorted in the first place. Is should be mentioned that many implementations of QuickSort use the first element of each subsequence as a partitioning pivot, which would impact its performance for nearly sorted inputs. Our implementation follows the recommendations of Sedgewick [11], using a randomized pivot element and using Insertion Sort for small subsequences.

The TimSort algorithm also fares relatively well and, being adaptive, should beat QuickSort for larger inputs, at least for inputs with a very small amount of modified elements. The SmoothSort algorithm, although elegant and with nice theoretic properties, is fairly complex to implement, exhibiting a poor performance in practice even for nearly 100% sorted inputs.

The Split-Merge strategy is the clear winner in this test. Since it uses the QuickSort algorithm to sort only the modified portion of the array, it incurs in only a small overhead for performing the splitting and merging operations, which are $O(n)$. The cost of this overhead is only made to bear when the input array is fairly well scrambled.

The second batch of tests benchmarked GPU sorting strategies, contrasting the performance of one CPU and two GPU implementations the Split-Merge strategy described in Sections IV-B, V-A and V-B with those of two standard GPU sort

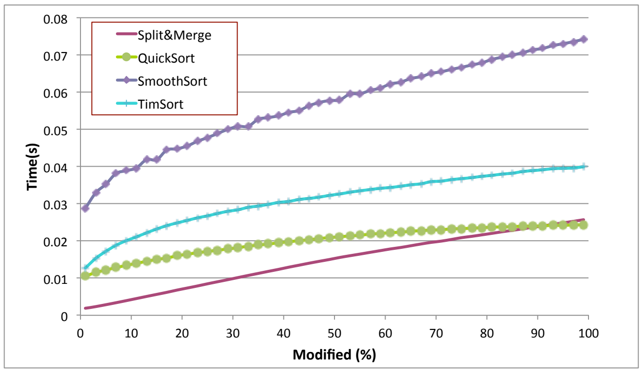

Fig. 3. Comparison of time taken by Algorithm 2 to reorder a nearly sorted collection of 256k integers using various Sort strategies as a function of the percentage of modified elements.

algorithms, i.e., Radix Sort and Bitonic Sort. In order to ease the comparison, we use the concept of *speedup*, which is the relative improvement of an algorithm of interest with respect to a reference algorithm. In particular, all charts for these tests use the performance of either the Radix or the Bitonic Sort as a reference, depending on which is faster. Thus, we define the speedup for an algorithm "X" over the reference algorithm as

$$speedup(X) \equiv \frac{min(time(\text{Bitonic}), time(\text{Radix}))}{time(X)}.$$

Figures 7 and 8 show speedup plots of of tests conducted in systems A and B, respectively. Due to the limited available space, only four test scenarios are shown for each system, corresponding to the use of keys with 20 and 32 bits for input batches of 64k and 1024k elements. Shorter keys are suitable for coarser spatial grids whereas longer keys encode hashes for finer grids. The different key lengths are relevant for this kind of tests because they influence the performance of the Radix Sort. For instance, 20-bit keys require $20/4 = 5$ steps, whereas 32-bit keys require $32/4 = 8$ steps. This is reflected in the results where the reference algorithm for all scenarios was the Radix Sort, except in the case of 32 bit keys and 64k elements, where that algorithm is outperformed by the Bitonic Sort. It should also be mentioned that all reference algorithms ran in almost constant time with respect to the modified rate. A plot of the various average times is shown in Figure 4.

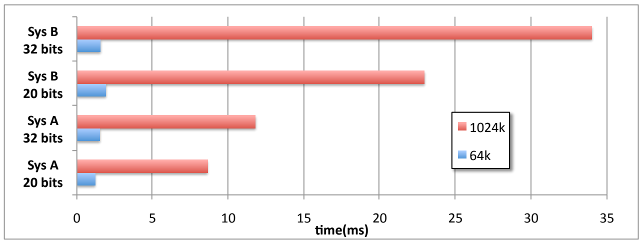

Fig. 4. Average times taken by the reference algorithms

It is interesting to observe that the performance of the GPU 1 implementation seems to exhibit a staircase pattern with increasingly larger plateaus. This is explained by the fact that the reference GPU algorithm used for sorting the modified subsequence always require with sizes that are powers of two, and so the input must be padded to the next larger such value.

An analysis of Figures 7 and 8 suggests that both GPU implementations of the Split and Merge strategy are clearly superior to the reference algorithms when the ratio of modified element is small to medium (5 to 30%) in all scenarios. They fare even better for larger inputs. Even the CPU implementation beats the reference GPU algorithms for small modified ratios in System B, which has a relatively slow GPU paired with a fast CPU. This kind of pairing also seems to favor GPU implementation 2 over 1, whereas in System A the differences between the two have been less evident. Absolute gains were also more pronounced in System B, with over 2× speedups for 32-bit scenarios within a modified rate up to 10%.

B. Simulation tests

Two particle simulation tests were built employing the aforementioned grid-based collision detection. These are called *Rough Sea* and *Gravity Effect*, and two frames of each simulation are shown in Figure 5.

The simulations were inspired on the *particles* demo included in NVidia's OpenCL SDK [16]. Both simulations were run on System A, using a 2M-cell grid, with 128 subdivisions by axis. Each simulation was run for 10,000 time steps, where rendering was done at every 3 steps. The Rough Sea simulation contains 512k particles and physics consists of a uniform gravity field which clusters particles on the bottom of the box and contact repulsion between particles and with the box walls. The Gravity Effect simulation contains 128k particles, where physics consisted mainly of two poles attracting the particles and an effect of repulsion between contacting particles or between particles and the containing box. A comparison showing

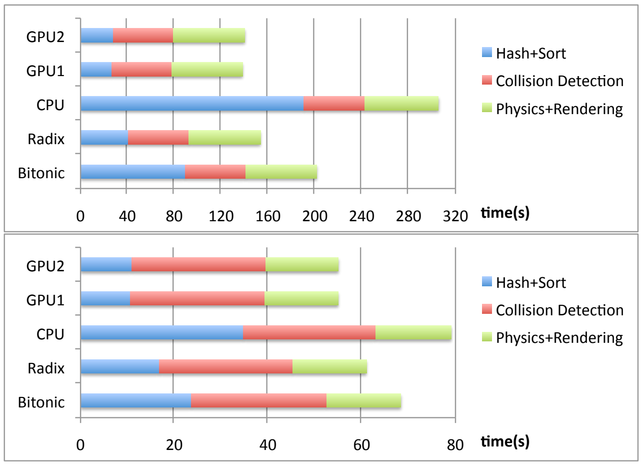

Fig. 6. Timings for physics simulation runs: Rough Sea (top) and Gravity Effect (bottom)

the times of the three main steps for each simulation is shown in Figure 6. As expected, the times spent in processing the collisions, physics and rendering were not affected by the

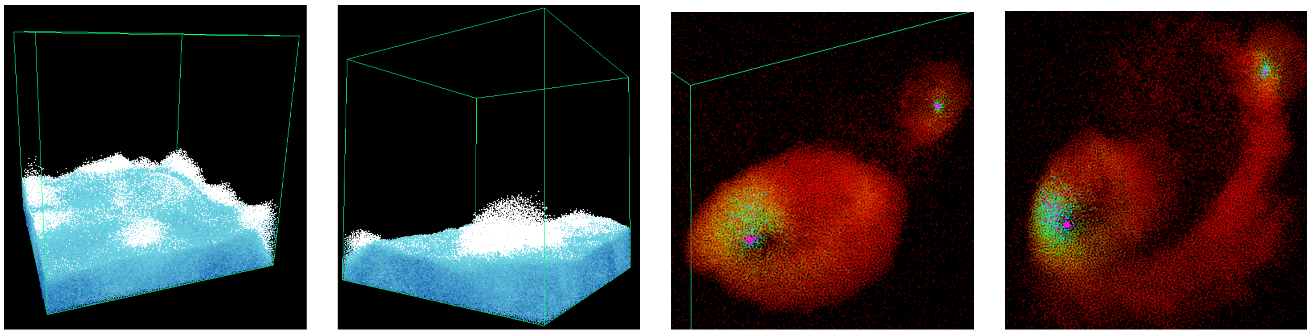

Fig. 5. Two frames from the "Rough Sea" (left) and "Gravity Effect" (right) simulations.

sorting strategy employed. In all tests, we notice that using either GPU Split-Merge strategy implementations improves the overall times by roughly 10% when compared with runs using Radix Sort, which fared better than Bitonic Sort in all cases. In the Rough Sea, GPU2 fared worse than GPU1, which was not expected, given that standalone results for System A using 32-bit keys with 512k particles (not shown in this paper) favored the former over the latter. This can be explained by the clustering of particles in the bottom half of the grid, which makes the intervals that must be sorted in the last phase of the algorithm unduly large.

VII. Final Remarks

The techniques described in this paper have been shown to be advantageous under some conditions. The empirical evidence shown in Section VI although necessarily limited, points to the usefulness of the Split-Merge strategy as an adaptive sort technique, especially when input data vary relatively slowly over time, and provided that the application has some way of distinguishing the modified portions of the input. This is exactly the case of particle simulations using spatial grids. Obviously, the net improvement observed for the application as a whole depends on the time spent in the sort phase. Also, the measurements were conducted for relatively few scenarios, and using our own implementations, and thus cannot be considered a complete proof of concept.

As a continuation of this work, we plan on experimenting with more modern GPU architectures. We are also interested in porting traditional adaptive sorting algorithm such as the TimSort to GPU.

References

[1] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Sliced data structure for particle-based simulations on GPUs," in *GRAPHITE*, 2007, pp. 55–62.

[2] T. Harada, M. Tanaka, S. Koshizuka, and Y. Kawaguchi, "Real-time coupling of fluids and rigid bodies," in *APCOM 2007-EPMESC XI (2007)*. University of Tokyo, 2007.

[3] Y. Dobashi, Y. Matsuda, T. Yamamoto, and T. Nishita, "A fast simulation method using overlapping grids for interactions between smoke and rigid objects," 2008.

[4] S. Green, "Particle simulation using cuda." NVIDIA - presentation packaged with CUDA Toolkit, 2010.

[5] M. C. Lin and S. Gottschalk, "Collision detection between geometric models: A survey," in *In Proc. of IMA Conference on Mathematics of Surfaces*, 1998, pp. 37–56.

[6] C. Ericson, *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann, Jan. 2005. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/1558607323

[7] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and R. Rowe, "Collision detection: A survey," in *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, Oct., pp. 4046–4051.

[8] T. Harada, "Real-Time rigid body simulation on GPUs," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, Aug. 2007, ch. 29. [Online]. Available: http://my.safaribooksonline.com/9780321545428/ch29

[9] S. Le Grand, "Broad-Phase collision detection with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, Aug. 2007, ch. 32.

[10] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, pp. 321–, Jul. 1961. [Online]. Available: http://doi.acm.org/10.1145/366622.366644

[11] R. Sedgewick, "Implementing quicksort programs," *Commun. ACM*, vol. 21, no. 10, pp. 847–857, Oct. 1978. [Online]. Available: http://doi.acm.org/10.1145/359619.359631

[12] D. R. Martin, "Sorting algorithm animations," http://www.sorting-algorithms.com/, 2007.

[13] T. Peters, "Timsort," 2002, accessed April 2013. [Online]. Available: http://svn.python.org/projects/python/trunk/Objects/listsort.txt

[14] E. W. Dijkstra, "Smoothsort, an alternative for sorting in situ." *Sci. Comput. Program.*, vol. 1, no. 3, pp. 223–233, 1982, errata: Science of Computer Programming 2(1): 85 (1982). [Online]. Available: http://dblp.uni-trier.de/db/journals/scp/scp1.html#Dijkstra82

[15] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, vol. 24, no. 4, pp. 441–476, Dec. 1992. [Online]. Available: http://doi.acm.org/10.1145/146370.146381

[16] NVIDIA, "Nvidia OpenCL SDK code samples," http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html, 2012.

[17] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[18] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161005

[19] G. E. Blelloch, "Prefix sums and their applications," Synthesis of Parallel Algorithms, Tech. Rep., 1990.

[20] ——, *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[21] T. Ottmann, "Parallel merging," ser. Lecture Notes in Computer Science - Advanced Algorithms & Data Structures, no. 15, 2006. [Online]. Available: http://electures.informatik.uni-freiburg.de/portal/download/3/6950/thm15%20-%20parallel%20merging.pdf

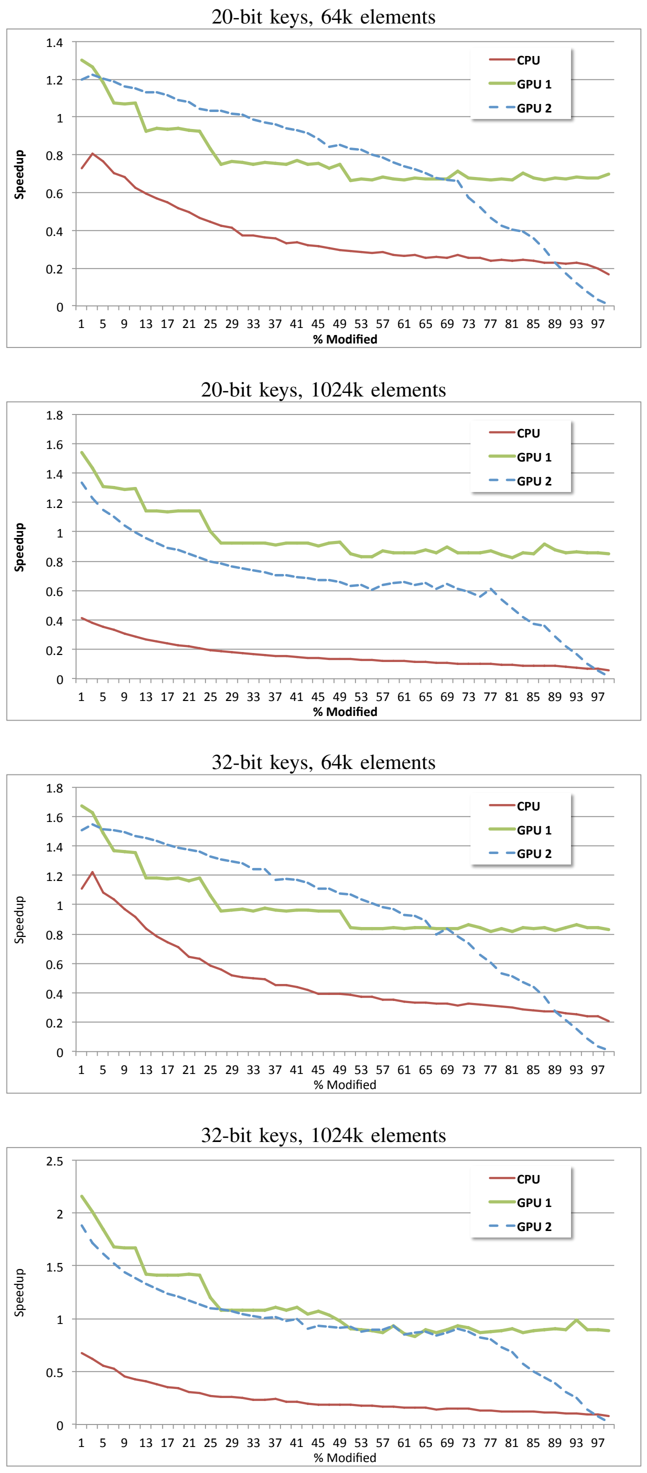

Fig. 7. Speedup charts for GPU sorting strategies ran on System A.

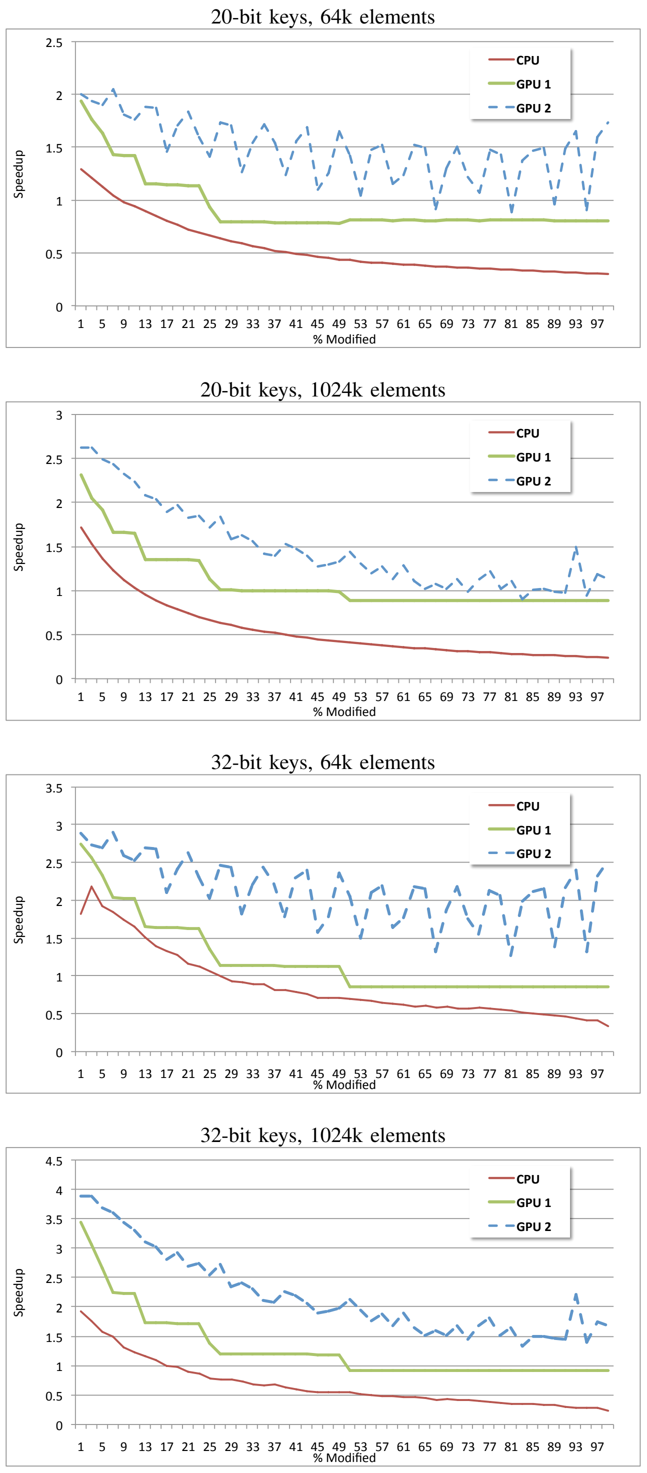

Fig. 8. Speedup charts for GPU sorting strategies ran on System B.