J Supercomput (2015) 71:3482–3499
DOI 10.1007/s11227-015-1445-5

SP-ChainMail: a GPU-based sparse parallel ChainMail algorithm for deforming medical volumes

Alejandro Rodríguez[1] · **Alejandro León**[1] · **Germán Arroyo**[1] · **José Miguel Mantas**[1]

Published online: 23 May 2015

Abstract ChainMail algorithm is a physically based deformation algorithm that has been successfully used in virtual surgery simulators, where time is a critical factor. In this paper, we present a parallel algorithm, based on ChainMail, and its efficient implementation that reduces the time required to compute deformations over large medical 3D datasets by means of modern GPU capabilities. We also present a 3D blocking scheme that reduces the amount of unnecessary processing threads. For this purpose, this paper describes a new parallel boolean reduction scheme, used to efficiently decide which blocks are computed. Finally, through an extensive analysis, we show the performance improvement achieved by our implementation of the proposed algorithm and the use of the proposed blocking scheme, due to the high spatial and temporal locality of our approach.

Keywords GPU programming · Stencil computation · Physically based deformation · Parallel algorithms

1 Introduction

Over the last few years, graphics processing units (GPUs) have been widely used to accelerate a huge variety of algorithms in different fields. This is due to the fact that modern GPUs are designed following a highly parallel single instruction, multiple data (SIMD) scheme, containing hundreds or thousands of processors and dedicated memory.

Many approaches for physically based deformation of medical volumetric models take advantage of these capabilities such as the parallel implementation of the finite

✉ Alejandro Rodríguez
alejandrora@ugr.es

[1] University of Granada, Granada, Spain

element method proposed by Comas et al. [2] or the parallel mass–spring system proposed by Georgii et al. [8], since they can be adapted to operate in parallel over a huge amount of data elements. The ChainMail algorithm, introduced by Gibson [10], is a two-stage physical deformation algorithm which, unlike other physically based deformation algorithms, follows a purely geometrical approach that is therefore capable of handling several orders of magnitude more elements. It has been successfully used to simulate surgical procedures, such as a vitrectomy [19] or arthroscopic knee surgery [9], where a low response time is a strong requirement.

In this paper, we present a parallel version of the ChainMail algorithm which efficiently handles deformations over large areas of the dataset. Our algorithm allows a parallel implementation of the tasks that are computationally intensive using the GPU, thus avoiding costly memory transfers to visualize the deformations since the rendering is also carried out using the GPU. Unlike previous approaches, our algorithm is capable of interleaving its two stages, allowing intermediate visualizations of the current state of the deformation. Therefore, our algorithm is able to provide a more interactive visual feedback.

We also propose a partitioning method that, taking into account the sparse nature of our algorithm, splits up the computation of the dataset elements into blocks that can be processed independently. This blocking method prevents the processing of blocks that do not require any computation, reducing the number of idle threads, thus decreasing the overall computation time.

Additionally, we present a novel parallel reduction approach that is limited to reduction of boolean sets, but improves the performance of the general parallel reduction approach.

This paper is organized as follows. In Sect. 2, previous related work is reviewed. Our parallel ChainMail algorithm is described in Sect. 3, which also includes a brief introduction to the original ChainMail algorithm, as well as details about the implementation. In Sect. 4, the blocking method is described and the required algorithms and data structures to efficiently handle the blocks are detailed. Also, the proposed boolean reduction mechanism is presented. In Sect. 5, we present an analysis of results, testing our approach under different blocking configurations and several datasets and comparing its performance against an optimized multithreaded implementation of the original ChainMail algorithm. We extend the analysis to several hardware configurations, demonstrating the portability and scalability of the blocking scheme and the benefits achieved by our approach. Finally, our conclusions are given in Sect. 6.

2 Related works

To take advantage of general-purpose computing on graphics processing units (GPGPU), it is necessary to map the algorithms to the graphics hardware, which is not always an easy task. Kirk et al. [11] presented an excellent introduction to massively parallel general-purpose computation using modern graphics hardware, compiling recent developments, common techniques and several practical examples. Due to the SIMD nature of modern GPUs, a common approach to perform parallel computation is the iterative stencil computation scheme [3]. This scheme consists of a sequence

of iterations over a given dataset, stored in a grid of cells. Each iteration performs local neighborhood computations to obtain new values for the cells. Examples of this approach are the stencil-based GPU algorithm proposed by Micikevicius [14] to perform 3D finite difference calculations, and the iterative parallel approach proposed by De la Asunción et al. [4] to simulate shallow water systems on the GPU.

The original ChainMail algorithm [10] has been used by many authors for medical applications, such as angioplasty simulation [12] , heterogeneous deformation of medical datasets [20] and generation of medical illustrations [13]. Unfortunately, interactivity is only achieved if the amount of affected elements is relatively small. Since the original ChainMail algorithm presents an important computational stage which is inherently sequential, a direct mapping to parallel platforms computation has a very limited impact on the performance.

A two-stage parallel approach based on the ChainMail algorithm was introduced by Rößler [17] and has been also used in medical applications by Fortmeier et al. [5,6]. This parallel approach achieves good performance for small deformations, but suffers from a high amount of idle computation when large deformations are applied, hurting the overall performance. Moreover, the visualization can only be performed after the whole deformation is completed, decreasing the visual feedback and interactivity during large deformations.

Unlike previous approaches, our algorithm handles both the propagation and the relaxation stage at the iteration level following a stencil computation scheme, allowing overlapping both stages to generate partial visualizations of the deformations. Moreover, the use of our blocking scheme avoids unnecessary computation, increasing the performance of the overall process.

Bandwidth and computation problems associated with the stencil computation approach have been widely studied. Many cache-based blocking schemes palliate bandwidth problems. An example is discussed in the work of Nguyen et al. [16], introducing a 3.5D spatial and temporal blocking scheme applied to the input grid into on-chip memory to optimize bandwidth bounded kernels. Brodtkorb et al. [1] proposed an early exit mechanism to avoid further computation of blocks marked as non-contributing in the previous iteration. Sætra [18] proposed methods to reduce the computational burden and required memory to perform stencil operations over sparse domains.

Our blocking scheme extends the work of Brodtkorb et al. [1] to efficiently handle the activation and deactivation of the blocks, further reducing the unnecessary computation performed in each iteration of our stencil computation.

3 SP-ChainMail

The original ChainMail algorithm [10] defines a mesh structure over the elements of the volumetric model. Each element is connected to its six adjacent neighbors. A deformation is handled by two separate stages: propagation stage and relaxation stage.

In the *propagation stage*, a valid spatial region is defined for each neighbor of a given element. While a neighbor remains within that region, the state is valid and no updates are needed. Since the regions are defined relative to the current position of

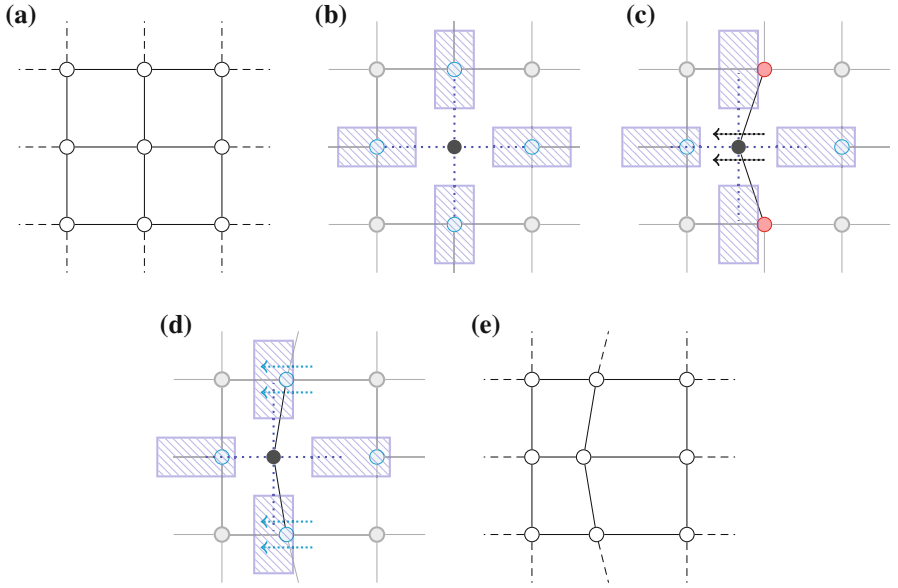

Fig. 1 2D depiction of an element deformation using the ChainMail algorithm. **a** Given an initial configuration, **b** the element defines valid regions for its neighbors, **c** when the element is displaced, it defines new valid regions, **d** the neighbors violating those new constraints are shifted to fulfill them. **e** The final stable state is reached when all the constraints are satisfied

each element, the valid regions for the neighbors of an element are displaced when that element is displaced. Due to this displacement, a neighbor may be outside the new valid region. If this happens, the neighbor is shifted to a new location to fulfill the constraint, as shown in Fig. 1. The shifting of a neighbor may, in turn, lead to new constraint violations and cause further displacement of elements, propagating the deformation through the mesh elements.

Once all the constraints are satisfied, the *relaxation stage* begins: each element is iteratively displaced towards its equilibrium position based on a midpoint calculation of the positions of its neighbors. Rigid, plastic and elastic behaviors can be achieved by tuning up the geometric constraints between elements and modifying the relaxation scheme, as described by Gibson [7].

The ChainMail algorithm is implemented using the CPU since the propagation stage of the algorithm is inherently sequential, and the deformed mesh must be transferred to the graphics device memory to perform the visualization. For large models, this memory transfer is expensive and impedes an interactive visualization of the applied deformations.

3.1 Sparse parallel ChainMail

In our approach, the volumetric model is arranged as a regular, structured 3D grid of cells. Each cell corresponds to an element of the ChainMail mesh. Hence, a cell stores the 3D position of its associated element and the connections with its neighbors.

Two copies of the grid are used following a Jacobi sweep scheme [16]: one grid is designated to stencil read operations and the other is designated to stencil write operations, swapping roles after each iteration. Both grids are stored in the device memory (dedicated memory of the GPU). Operations over cells corresponding to the propagation and relaxation stages are performed following a stencil computation approach:

- In the *propagation stage* the original propagation mechanism is inverted as explained by Rößler [17], adapting it to follow an iterative stencil-based approach: for each cell, if a neighbor has been displaced on the previous iteration, the new constraint is checked. If the constraint is not satisfied, the element is shifted to meet the existing geometric constraint with its neighbor. This process is repeated iteratively until all the constraints are satisfied.
- In the *relaxation stage*, a minimization process is applied based on the elastic and plastic properties of the model as explained by Gibson [7]. This energy minimization process also follows an iterative stencil-based approach, since each cell updates its position as a result of a computation regarding the current positions of its neighbors.

The computation performed for each cell during a propagation iteration, as well as during a relaxation iteration, is independent of the computation performed for the rest of the cells, allowing a parallel computation of each iteration.

Unlike the previous solutions, we introduce a mechanism to handle the stages at the iteration level. This mechanism requires adding a control flag for each cell. This flag tracks whether the cell has already been reached by the current propagation front. Therefore, when a new external deformation is applied to a cell, it is flagged as *reached* and the rest of the cells are flagged as *not reached*. During subsequent propagation iterations, when an element is reached by the propagation front, it is flagged as *reached*.

After each iteration of the propagation stage, this flag allows to identify the cells that have already been reached by the propagation front. If a cell and its neighbors have already been reached, the cell is ready to perform the relaxation stage. Therefore, this flag allows overlapping the propagation and relaxation stages by alternating propagation and relaxation iterations. This overlapping mechanism allows to visualize partial results of the deformations and, since the updated data after any propagation or relaxation iteration is already present in the device memory, no memory transfers are required, allowing a more interactive visual feedback. An overview of the proposed algorithm integrated in a virtual surgery system is depicted in Fig. 2.

3.2 Parallel implementation

In our approach, all the computationally intensive tasks are executed in parallel using the GPU. Hence, the 3D dataset is loaded into the device global memory as an array of cells. Each cell stores the following information:

- *Element data*: position and constraint values.

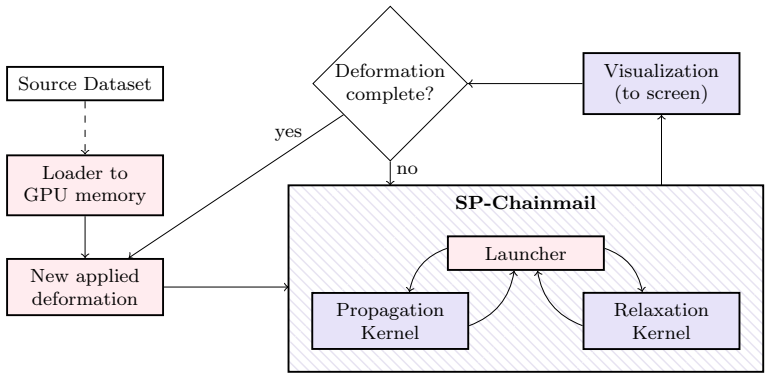

Fig. 2 Overview of the simulation system. The SP-ChainMail algorithm runs in parallel on the GPU, allowing partial visualizations of the deformations. *Blue stages* are carried out using the GPU, and *red stages* are carried out using the CPU

- *Neighbors flags*: a set of six flags indicating whether the element is connected or not with each of its neighbors (its six surrounding cells in the grid).
- *Activity flag*: a flag indicating if the element has been displaced in the preceding propagation iteration.
- *Reach flag*: a flag indicating if the element has already been reached by the current propagation.

To cope with the Jacobi sweep scheme, we duplicate the whole array supporting read and write operations. The current read array will be referred to as *global read array*, and the current write array will be referred to as *global write array*.

3.2.1 Propagation stage

The propagation stage is implemented as a GPU kernel that is iteratively invoked. Each kernel invocation computes a single iteration of the propagation, generating one thread per cell. The kernel is described in Algorithm 1:

1. The cell data corresponding to the current thread is read from the global read array (lines 2–3).
2. For each neighbor, the following condition is checked (lines 5–6): the neighbor has been shifted in the previous propagation iteration and the new restrictions are violated.
3. If this condition is met, the current element is shifted to fulfill the new constraints and it is flagged as *reached* and *active* (lines 7–9).
4. Otherwise, the current element is flagged as *inactive* (line 4).
5. Finally, the cell data are written to the global write array (line 12).

If no elements are shifted during the kernel invocation, the propagation stage finishes and no more propagation iterations are needed.

```
1  propagation_kernel (Cell readArray[], Cell
       writeArray[])
2      Integer id = get_thread_id();
3      Cell current = readArray[id];
4      current.activityFlag = False;
5      FOREACH neighbor IN activeNeighbors(current)
6          IF ( (neighbor.activityFlag == True) AND
       (restrictionsNotSatisfied(current, neighbor))
       )
7              relocate(current);
8              current.activityFlag = True;
9              current.reachFlag = True;
10         ENDIF
11     ENDFOREACH
12     writeArray[id] = current;
13 END
```

Algorithm 1 Pseudo-code of the propagation kernel. During the kernel invocation, each instance of this kernel operates over a single cell, writing the resulting updated cell to the current global write array

3.2.2 Relaxation stage

The relaxation stage is also implemented as a GPU kernel that is iteratively invoked. Each invocation of the kernel computes a single relaxation iteration, generating one thread per cell. The kernel is described in Algorithm 2:

1. The cell data corresponding to the current thread is read from the global read array (lines 2–3).
2. The following condition is checked (line 4): the current element has already been reached, but it is not active.
3. If this condition is met and all the neighbors of the current element have already been reached (lines 5–11), the relaxation process is applied to the current element (line 12).
4. Finally, the cell data are written to the global write array (line 15).

If none of the elements is shifted during a relaxation iteration and the propagation stage has been already completed, the relaxation stage also finishes and the deformation is completed.

After an invocation of any of these kernels, the global arrays switch their roles, allowing the next kernel invocation to read from the updated array. Since the relaxation kernel only affects the elements already reached by the propagation, excluding those belonging to the current propagation front, both kernels can be interleaved. The visualization of the current state of the deformation is also possible by accessing the array data updated by the latest iteration.

Some details regarding our implementation have been omitted for the sake of clarity. To improve the efficiency of the GPU kernels, we have adopted the following strategies:

– The *foreach* loops are completely unrolled.
– The GPU shared memory is used to optimize the access to neighboring cells.

```
relaxation_kernel (Cell readArray[], Cell
    writeArray[])
    Integer id = get_thread_id();
    Cell current = readArray[id];
    IF ( (current.reachFlag == True) AND (current
    .activityFlag == False ) )
        Boolean continue = True;
        FOREACH neighbor IN activeNeighbors(
    current)
            IF (neighbor.reachFlag == False)
                continue = False;
            ENDIF
        ENDFOREACH
        IF (continue == True)
            applyRelaxationFunction(current);
        ENDIF
    ENDIF
    writeArray[id] = current;
END
```

Algorithm 2 Pseudo-code of the relaxation kernel. During the kernel invocation, each instance of this kernel operates over a single cell, writing the resulting updated cell to the current global write array

– The actual data of the cells are stored in a structure-of-arrays fashion, more amenable to the regular memory access patterns of the kernels.

4 Computational blocking method

Our stencil approach presents a high spatial and temporal locality of the computational burden, since the deformations applied to the model propagate iteratively through the regular grid following a wavefront pattern. This leads to a highly sparse computation in the propagation stage, resulting in a high amount of unnecessary computation.

This unnecessary computation is produced because many of the elements may have already been shifted in a previous iteration or have not yet been reached by the current propagation. A less severe sparse computation is also present in the relaxation stage because of the same reason. Due to this sparse computation, many of the launched threads would be idle, wasting GPU resources, since these threads also need to read from the device global memory to compute the data.

Since our solution follows an iterative stencil computation approach, we can introduce a blocking method to reduce the number of idle threads, optimizing the usage of the computation power offered by the GPU.

For this purpose, the computational domain is divided into blocks that can be computed independently. The storage of the dataset in the device memory remains the same, but each block is handled by an independent kernel launch instead of a single kernel launch over the whole dataset.

To maintain this structure, we store the corresponding 3D offset for each block. During a kernel launch for a particular block, the kernel receives this offset information to access the data of the cells in that block.

After each iteration, the blocks are flagged as *active* or *inactive*. *Active* means that the block may require further computation in the next iteration, while *inactive* means that the block will not need further computation in the following iteration. These flags allow launching the kernel only over active blocks to avoid unnecessary computation. This blocking scheme is applied to both stages of our algorithm in an efficient way as explained in the following subsections.

4.1 Efficient activation and deactivation of blocks

To handle the activation and deactivation of blocks, we extend the solution proposed by Brodtkorb et al. [1], which involves the use of an auxiliary boolean buffer to indicate whether a block requires computation in the next iteration or not. In our approach, we use several of these buffers, referred to as *boolean maps*, to update and control the state of the blocks.

Each boolean map is stored as a global array on device memory containing one binary flag per block in the partition. A first boolean map is associated with the propagation stage. A second boolean map is associated with the relaxation stage.

In each iteration, for any of the two stages, the blocks that need to be computed in the next iteration are flagged as active in the corresponding boolean map. A new condition test, added to the end of the kernels code, decides whether a block requires further computation or not by checking if any element in the block has been updated. An element is considered updated by a propagation iteration if it has been reached by the propagation front. An element is considered updated by a relaxation iteration if it has been displaced by the relaxation function. On the other hand, if none of the elements in a block have been updated during the current iteration, the block is flagged as inactive. Figure 3 shows a 2D illustration of this mechanism.

4.1.1 Activation of neighboring blocks

When the propagation front or the relaxation process reaches the border of a block, the neighboring block must be activated. Six additional boolean maps are defined, each one associated with one of the borders for all the blocks.

Therefore, if an element belonging to the border of a block is updated in the current iteration, the position of that block in the boolean map corresponding to that border is set as *active*.

In the host memory (main memory), two lists of active blocks are maintained. At the end of each iteration, for any of the two stages, the corresponding boolean maps are copied to the host memory and the corresponding list is updated using the boolean maps as look-up tables. At the beginning of the next iteration, only the blocks indexed in the corresponding list are processed by the kernel.

Figure 4 presents the steps and memory accesses during an iteration of the algorithm. Notice that the dataset is always stored in the device memory and operated from the GPU, and only the boolean maps are transferred to the host memory.

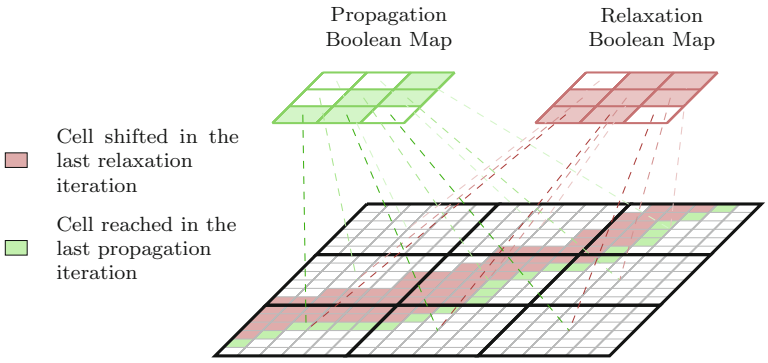

Fig. 3 2D simplification of the boolean map mechanism. Blocks containing cells reached in the last propagation iteration are flagged in the propagation boolean map, and blocks containing cells shifted in the last relaxation iteration are flagged in the relaxation boolean map

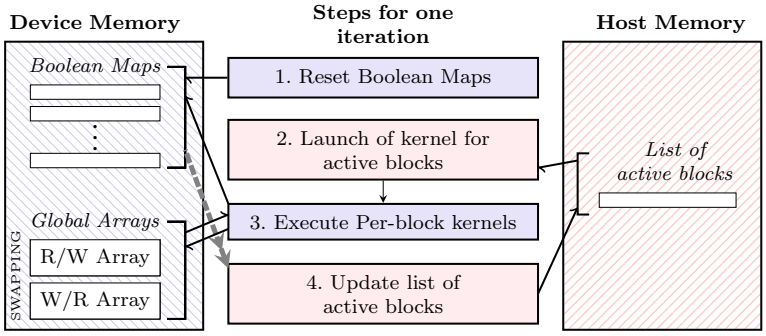

Fig. 4 Steps and memory accesses during an iteration. The *blue steps* (steps 1 and 3) run on the GPU, while the *red steps* (steps 2 and 4) run on the CPU. The only memory transfer between device memory and host memory is performed in step 4 to update the active blocks list

4.2 Parallel boolean reduction

As mentioned earlier, the boolean maps are updated by the kernels, but, since each launched thread handles only one cell, it is necessary to perform a gathering process regarding each block. Instead of a parallel reduction approach as in [1] and [18], we propose a novel two-step parallel boolean reduction (PBR) mechanism:

1. All the flags of the boolean maps are set as *inactive* before launching the kernels corresponding to the active blocks, assuming that none of the blocks will need further computation in the next iteration.
2. The kernels corresponding to the active blocks are launched. If an element of a block is updated, a write operation is performed to set as *active* the position of that block in the corresponding boolean map.

Although the concurrent writing of several threads to the same variable does not guarantee the integrity of data, in this case all the threads write the same value. This fact ensures the final state of the boolean values while avoiding the additional latency

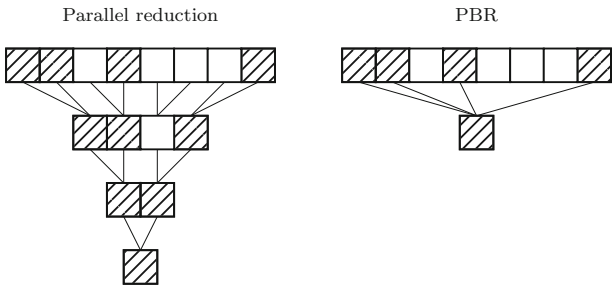

Fig. 5 *Left* parallel reduction of eight boolean values using the binary OR operation. *Right* parallel boolean reduction (PBR) of the same eight boolean values

introduced by a parallel reduction approach. A depiction of both approaches operating over the same set of values is shown in Fig. 5. A performance comparison of both methods is presented in the next section.

5 Experiments and results

To demonstrate the benefits of the proposed methods, several tests have been conducted using different hardware configurations to also evaluate the portability and scalability of the proposed blocking scheme. Three hardware configurations have been used:

- *GTS-250* configuration: Intel Core i3-530 2.93 GHz, 4 GB RAM, Nvidia GeForce GTS 250 (Tesla microarchitecture, 128 cores) with 1 GB of video memory GDDR3. OpenCL 1.1 driver included in CUDA 6.
- *R9-270X* configuration: Intel Core i5-3570 3.4 GHz, 8 GB RAM, AMD Radeon R9 270X (1280 cores) with 2 GB of video memory GDDR5. OpenCL 1.2 driver.
- *GTX-670* configuration: Intel Core i7-3770 3.4 GHz, 16 GB RAM, Nvidia GeForce GTX 670 (Kepler microarchitecture, 1344 cores) with 2 GB of video memory GDDR 5. OpenCL 1.1 driver included in CUDA 6.

Two different datasets have been used for the tests. The first dataset, referred to as *Cube dataset*, is a synthetic regular 3D cube, consisting of $96 \times 96 \times 96$ elements. The second dataset, referred to as *Leg dataset* is a section of a leg from the Visible Human Project of the National Library of Medicine (see Fig. 6), consisting of $160 \times 160 \times 160$ elements.

5.1 SP-ChainMail performance

To evaluate the performance of our approach, the SP-ChainMail algorithm has been implemented, together with the blocking scheme, using OpenCL [15], integrating it into a virtual surgery system prototype.

The original ChainMail algorithm [10] has also been implemented as a reference. The propagation stage of the original algorithm is inherently sequential and cannot be parallelized for multicore processors, but the relaxation stage has been parallelized

Fig. 6 *Left* the Leg dataset, used in the performance tests. *Right* the Leg dataset deformed by the sparse parallel ChainMail algorithm, using the developed virtual surgery system

using OpenMP by dividing the ChainMail elements in balanced groups and assigning the computation of each group to one thread.

A deformation has been applied to each dataset, causing a propagation relaxation through the whole dataset affecting all the elements. For the Cube dataset, the SP-ChainMail algorithm took 285 propagation iterations and 468 relaxation iterations until a completely stable configuration was reached. The original ChainMail algorithm also required 468 relaxation iterations after the propagation (not measurable in iterations). For the Leg dataset the SP-ChainMail algorithm took 477 propagation iterations and 788 relaxation iterations. The original ChainMail algorithm also required 788 relaxation iterations after the propagation. Each test has been repeated five times, although no noticeable differences were encountered through the different executions due to the deterministic behavior of the algorithms. The results presented here report the average of the measured times.

The original ChainMail was tested only on the GTX-670 configuration, our most modern hardware configuration. Using eight threads for the relaxation computation (grouping the elements into 8 groups), the stable state was reached after 10,648 ms for the Cube dataset and 49,283 ms for the Leg dataset.

We tested the SP-ChainMail implementation on both datasets using different block sizes for the blocking scheme, resulting in different rates of reduction on the total number of launched GPU threads. As can be seen in Table 1, a smaller block size always implies a higher reduction in the number of launched threads, which is expected since the smaller block sizes lead to a finer adjustment of the active blocks to the actual propagation front.

Table 2 shows the measured times using the Cube dataset and Table 3 shows the measured times using the Leg dataset. The times reported represent the time taken to reach the stable state for the same applied deformation to the dataset. The speed-up

Table 1 Thread launch reduction achieved using different block sizes

Block size	Cube dataset			Leg dataset		
	# of blocks	Launched threads	Thread launch reduction (%)	# of blocks	Launched threads	Thread launch reduction (%)
No blocks	–	666,206,208	–	–	5,181,440,000	–
32 × 32 × 32	27	234,553,344	64.79	125	1,018,888,192	80.33
32 × 16 × 16	108	148,873,216	77.65	500	639,262,720	87.66
16 × 16 × 16	216	107,683,840	83.83	1000	465,821,696	91.01
32 × 8 × 8	432	103,868,416	84.40	2000	451,835,904	91.27
16 × 8 × 8	864	67,950,592	89.80	4000	297,355,264	94.26
8 × 8 × 8	1728	49,827,840	92.52	8000	219,824,128	95.75

Table 2 Measured times of our SP-ChainMail implementation using the Cube dataset

Block size	GTS-250		R9-270X		GTX-670	
	Time (ms)	Speed-up	Time (ms)	Speed-up	Time (ms)	Speed-up
No blocks	8159	1.31×	1131	9.41×	1545	6.89×
32 × 32 × 32	4293	2.48×	693	**15.36**×	705	15.10×
32 × 16 × 16	3476	**3.06**×	815	13.06×	635	16.76×
16 × 16 × 16	3720	2.86×	1015	10.49×	598	**17.80**×
32 × 8 × 8	3923	2.71×	1484	7.17×	893	11.92×
16 × 8 × 8	4363	2.44×	1973	5.39×	1064	10.01×
8 × 8 × 8	5668	1.87×	2678	3.97×	1443	7.37×

Speed-up factors relative to the original ChainMail are also shown, highlighting the highest achieved speed-up for each configuration

with respect to the original ChainMail (running on the GTX-670 configuration) is also reported in both tables.

The tests reveal that the SP-ChainMail outperforms the original ChainMail even using relatively old GPUs, achieving notable speed-up factors higher than 20× when using a modern GPU. Interestingly, the results show that smaller blocks do not always lead to a higher speed-up, although the number of launched threads is smaller. This is due to the fact that the smaller kernel launches do not create enough parallel threads to fully hide the memory access latency, and this overhead, added to the overhead of managing more kernel launches, gradually decimates the gain of the reduced computation load.

5.2 Blocking method portability

For the GTS-250 configuration, any of the tested block sizes leads to a significant speed-up with respect to the non-partitioned case (i.e., the SP-ChainMail without using the blocking scheme), with the speed-up factor being higher when using the Leg dataset, since the thread launch reduction is higher. As already mentioned, the gain is

Table 3 Measured times of our SP-ChainMail implementation using the Leg dataset

Block size	GTS-250		R9-270X		GTX-670	
	Time (ms)	Speed-up	Time (ms)	Speed-up	Time (ms)	Speed-up
No blocks	61,271	0.80×	7601	6.48×	11448	4.30×
32 × 32 × 32	20,658	2.39×	2683	**18.36**×	2810	17.53×
32 × 16 × 16	16,130	**3.05**×	3207	15.36×	2428	20.29×
16 × 16 × 16	17,261	2.85×	3910	12.60×	2243	**21.97**×
32 × 8 × 8	18,128	2.72×	5742	8.58×	3438	14.33×
16 × 8 × 8	19,030	2.59×	7718	6.39×	4207	11.71×
8 × 8 × 8	25,291	1.95×	11,240	4.38×	5960	8.27×

Speed-up factors relative to the original ChainMail are also shown, highlighting the highest achieved speed-up for each configuration

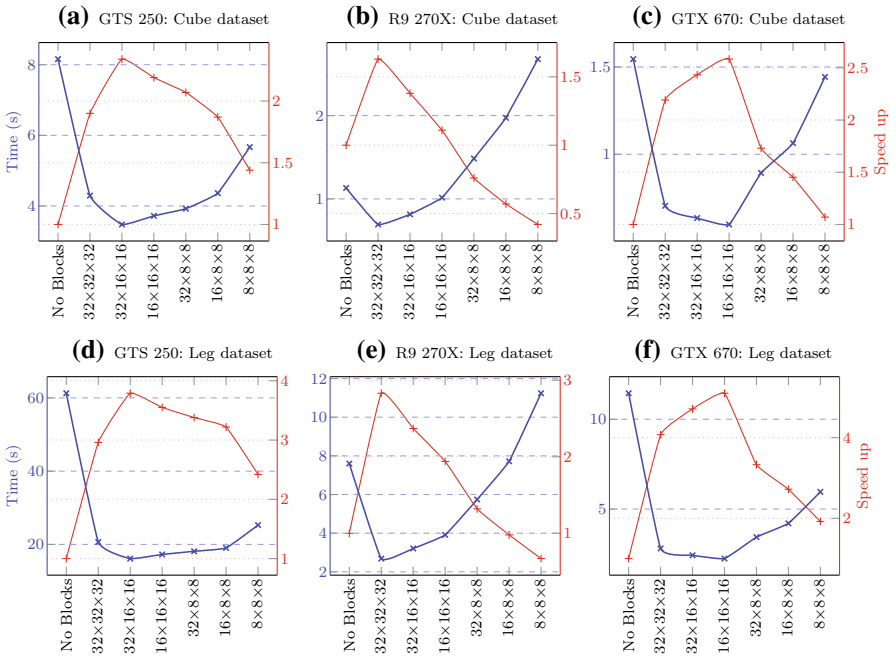

Fig. 7 Plots showing the time reduction and speed-up factor measured for the different block sizes with respect to the non-partitioned SP-ChainMail case. The *top row* shows the results using the Cube dataset. The *bottom row* shows the results using the Leg dataset, **a** GTS 250: cube dataset, **b** R9 270X: cube dataset, **c** GTX 670: cube dataset, **d** GTS 250: leg dataset, **e** R9 270X: leg dataset, **f** GTX 670: leg dataset

gradually decimated as the block size is reduced, due to the added overhead, as can be seen in Fig. 7a, d.

R9-270X and GTX-670 configurations exhibit a similar behavior (Fig. 7b, c, e and f) but, since the more recent GPUs present in those configurations have a much higher amount of stream processing units, they require an even higher amount of parallel threads to hide memory latency, and the smaller block sizes cannot even fully

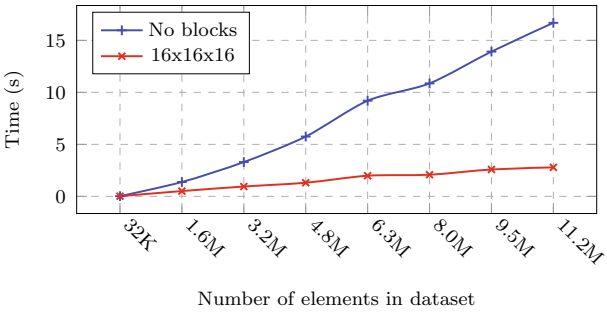

Fig. 8 Measured times of the scalability test, using the GTX-670 configuration

populate the GPU cores, leading to a loss of effective computation power. This loss is most severe in the case of the R9-270X configuration, on which the use of small block sizes even yields a worse performance than the non-partitioned case.

Despite this effect, the use of a reasonable block size (which depends on the particular GPU architecture) leads to a noticeable speed-up using any of the three configurations, showing the portability of the proposed blocking method and the performance gain obtained through its use.

5.3 Scalability test

Notice that the previous tests on the Leg dataset achieve a higher speed-up than their counterparts using the Cube dataset, suggesting a good scalability of the blocking method regarding the dataset size. To further analyze the scalability of our blocking method with respect to the dataset size, a second test using synthetic regular datasets, with dimensions ranging from $32 \times 32 \times 32$ to $224 \times 224 \times 224$, was performed.

The most modern configuration (the GTX-670 configuration) was used to perform this test. A deformation affecting all the elements of the dataset was applied, measuring the propagation time without using the blocking method and measuring the propagation time of the same deformation using a block size of $16 \times 16 \times 16$, which achieved the best performance gain on the GTX-670 configuration.

The measured times corresponding to this second test, presented in Fig. 8, show a significant reduction of the propagation time for all the tested dataset sizes, and they also show a good scalability of the proposed method since the speed-up factor, shown in Fig. 9, also increases when increasing the dataset size.

5.4 Memory requirements

5.4.1 SP-ChainMail memory requirements

The memory requirements of our SP-ChainMail algorithm (corresponding to the Global Arrays in Fig. 4) scale linearly with the number of elements in the input dataset.

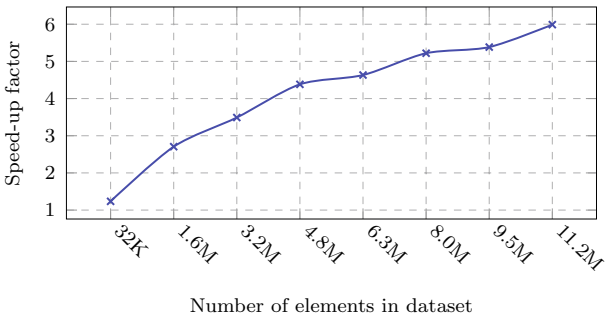

Fig. 9 speed-up factor achieved for the increasing dataset size of the scalability test, using the GTX-670 configuration

For each element, 30 bytes of device memory are required, which leads to a total amount of 480 MB for an input dataset of $256 \times 256 \times 256$ elements, and a total amount of 3.75 GB for an input dataset of $512 \times 512 \times 512$ elements, an amount currently offered only by high-end GPUs. However, this limitation is not reached in most scenarios, such as virtual surgery applications, since the simulation is usually performed on a sub-region of the dataset, and SP-ChainMail information would only be generated for the elements of the sub-region in those cases.

5.4.2 Blocking method memory requirements

The blocking method has very low host and device memory requirements.

In host memory (list of active blocks in Fig. 4), 64 bytes are required per block. In device memory, only 8 bytes are required per block.

In our most memory-demanding test (the Leg dataset with a $8 \times 8 \times 8$ block size, generating 8000 blocks in the partition) required 500 KB of device memory and 62.5 KB of device memory. As mentioned in Sect. 4.1.1, only the boolean maps are transferred from device memory to host memory at the end of each iteration. Even in our most memory demanding test, this transfer consumes <1 ms, which is a negligible overhead considering the achieved gain.

5.5 PBR performance test

A performance test comparing the proposed parallel boolean reduction (PBR) algorithm with a general parallel reduction algorithm was conducted. Both algorithms were applied to reduce several arrays of boolean elements of a wide range of sizes.

The measured times of both algorithms using the GTX-670 configuration are shown in Fig. 10. The PBR algorithm shows a better performance for all the array sizes, since less read/write operations are needed and no synchronization steps are required.

6 Conclusions and future work

In this work, we have presented a sparse parallel ChainMail algorithm. The proposed algorithm has been implemented and integrated into a virtual surgery system, allowing

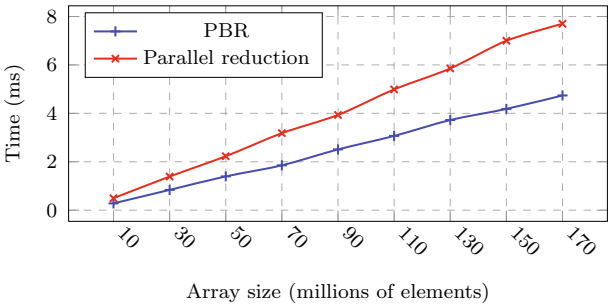

Fig. 10 Comparison of the times required by the parallel reduction algorithm and the PBR algorithm to perform a reduction over several boolean arrays, using the GTX-670 configuration

an interactive visual feedback during the manipulation of large volumetric models. Following a stencil computation approach, our algorithm adapts to the modern GPU computation paradigm.

We have proposed and implemented a 3D blocking method to deal with the sparse nature of the SP-ChainMail computation, drastically reducing the amount of idle GPU threads created.

A novel parallel boolean reduction mechanism has been used to efficiently handle the activation and deactivation of blocks. This reduction approach has been proven faster than a generic parallel reduction approach and can be used in any context in which the reduced value has a boolean nature, i.e., there are only two possible output values.

The tests conducted in this work show that our implementation considerably outperforms a parallel multithreaded implementation of the original ChainMail algorithm, and our blocking method effectively reduces the computation time required for the deformations, enhancing the interactivity of the simulation system. The tests also show a good portability and scalability of the blocking scheme, which increases its effectiveness as the dataset size increases, while the required additional memory is negligible.

As future lines of research, we intend to include an auto-tuning mechanism to determine the optimal block size automatically for each hardware and software configuration. Another interesting future line of work is the generalization and further testing of the blocking scheme for stencil computation approaches. Moreover, it would be interesting to test the use of dynamic parallelism to perform the handling and launching of the blocks directly from the GPU.

Acknowledgments This work was supported by the "Formación de Profesorado Universitario, Plan Propio de Investigación" program of the University of Granada. This work was also supported by the project TIN2014-60956-R of the Spanish Ministry of Economy and Competitiveness. JMM acknowledges the Spanish MINECO project MTM2014-52056-P.

References

1. Brodtkorb AR, Sætra ML, Altinakar M (2012) Efficient shallow water simulations on GPUs: implementation, visualization, verification, and validation. Comput Fluids 55:1–12

2. Comas O, Taylor ZA, Allard J, Ourselin S, Cotin S, Passenger J (2008) Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA. Biomedical simulation. Springer, London, United Kingdom, pp 28–39
3. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing. IEEE Press, Austin, Texas, p 4
4. De La Asunción M, Mantas JM, Castro MJ (2011) Simulation of one-layer shallow water systems on multicore and CUDA architectures. J Supercomput 58(2):206–214
5. Fortmeier D, Mastmeyer A, Handels H (2013) Image-based palpation simulation with soft tissue deformations using chainmail on the GPU. Bildverarbeitung für die Medizin 2013. Springer, Heidelberg, Germany, pp 140–145
6. Fortmeier D, Mastmeyer A, Handels H (2014) An image-based multiproxy palpation algorithm for patient-specific VR-simulation. Stud Health Technol Inform 196:107
7. Frisken-Gibson SF (1999) Using linked volumes to model object collisions, deformation, cutting, carving, and joining. IEEE Trans Vis Comput Graph 5(4):333–348
8. Georgii J, Echtler F, Westermann R (2005) Interactive simulation of deformable bodies on GPUs. In: SimVis, pp 247–258
9. Gibson S, Samosky J, Mor A, Fyock C, Grimson E, Kanade T, Kikinis R, Lauer H, McKenzie N, Nakajima S et al (1997) Simulating arthroscopic knee surgery using volumetric object representations, real-time volume rendering and haptic feedback. CVRMed-MRCAS'97. Springer, Grenoble, France pp 367–378
10. Gibson SF (1997) 3D ChainMail: a fast algorithm for deforming volumetric objects. In: Proceedings of the 1997 symposium on interactive 3D graphics. ACM, New York pp 149-ff
11. Kirk DB, Wen-mei WH (2012) Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, San Francisco, California
12. Le Fol T, Acosta-Tamayo O, Lucas A, Haigron P (2007) Angioplasty simulation using ChainMail method. In: Medical imaging, pp 65092X–65092X. International Society for Optics and Photonics, Bellingham
13. Mensmann J, Ropinski T, Hinrichs K (2008) Interactive cutting operations for generating anatomical illustrations from volumetric data sets
14. Micikevicius P (2009) 3D finite difference computation on GPUs using CUDA. In: Proceedings of 2nd workshop on general purpose processing on graphics processing units, pp 79–84. ACM, New York
15. Munshi A et al (2009) The OpenCL specification. Khronos OpenCL Work Group 1:l1–15
16. Nguyen A, Satish N, Chhugani J, Kim C, Dubey P (2010) 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis. IEEE Computer Society, Washington, DC pp 1–13
17. Rößler F, Wolff T, Ertl T (2008) Direct GPU-based volume deformation. In: Proceedings of Curac, pp 65–68
18. Sætra M (2013) Shallow water simulation on GPUs for sparse domains. Numerical mathematics and advanced applications 2011. Springer, Leicester, United Kingdom, pp 673–680
19. Schill MA, Gibson SF, Bender HJ, Männer R (1998) Biomechanical simulation of the vitreous humor in the eye using an enhanced chainmail algorithm. Medical image computing and computer-assisted intervention. Springer, Cambridge, Massachusetts, pp 679–687
20. Schulze F, Bühler K, Hadwiger M (2007) Interactive deformation and visualization of large volume datasets. In: GRAPP (AS/IE), pp 39–46. Citeseer