Arab J Sci Eng (2014) 39:7907–7921
DOI 10.1007/s13369-014-1406-y

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

A Generic Paradigm for Accelerating Laplacian-Based Mesh Smoothing on the GPU

Gang Mei · John C. Tipper · Nengxiong Xu

Received: 4 February 2014 / Accepted: 30 May 2014 / Published online: 12 October 2014

Abstract General purpose computing on Graphics Processor Units (GPGPU) can significantly reduce computational cost by performing massively parallel computing. The benefits of GPGPU have been exploited in mesh generation and mesh optimization. Mesh smoothing is one of the most popular approaches that are capable of improving mesh quality by repositioning nodes in meshes without altering the topology of meshes. In this paper, specifically aiming at Laplacian-based mesh smoothing, a generic paradigm for exploiting the power of GPU-acceleration is developed to improve the computational efficiency. The data layouts for representing mesh data structures on the GPU are first discussed; and then a practical solution to dealing with the data dependencies in the iterative smoothing procedure is introduced. Two forms of iteration in Laplacian smoothing (LS) are also analyzed, including the form that needs to swap intermediate nodal coordinates and the other form that does not swap data. In addition, the standard LS is implemented to demonstrate the effectiveness of the presented paradigm. Experimental results show that: on single and double precision, the GPU implementations developed using the data structures represented by aligned array-of-structures layout achieve the best efficiency. It is also demonstrated that the form that needs to swap intermediate nodal coordinates is always slower than the one that does not swap data.

G. Mei (✉) · N. Xu
School of Engineering and Technology, China University of Geosciences (Beijing), Beijing 100083, China
e-mail: gangmeiphd@gmail.com

G. Mei · J. C. Tipper
Institute of Earth and Environmental Science, University of Freiburg, Albertstr. 23B, 79104 Freiburg, Germany

Keywords GPU · Mesh smoothing · Iteration · Data layout · Data dependencies

الخلاصة

يمكن للحوسبة ذات الأغراض العامة على وحدات معالج الرسومات أن تقلل إلى حد كبير من التكلفة الحسابية عن طريق أداء الحوسبة المتوازية. وقد تم استغلال فوائد الحوسبة ذات الأغراض العامة على وحدات معالج الرسومات في توليد وتحسين الشبكة. ويُعد تمهيد الشبكة أحدَ الأساليب الأكثر شعبية القادرة على تحسين نوعية الشبكة عن طريق إعادة تموضع العقد في الشبكات دون تغيير طوبولوجيا الشبكات.

وفي هذه الورقة العلمية ، ولهدف تمهيد الشبكة المستند إلى لابلاسيان على وجه التحديد تم تطوير نموذج عام لاستغلال قوة تسارع وحدات معالج الرسومات لتحسين كفاءة الحسابية. ونوقشت أولا تخطيطات البيانات لتمثيل هياكل البيانات على شبكة وحدات معالج الرسومات. وتم بعد ذلك إدخال حل عملي للتعامل مع تبعيات البيانات في عملية التمهيد التكرارية. وتم أيضا تحليل اثنين من أشكال التكرار في تمهيد لابلاسيان ، بما في ذلك النموذج الذي يحتاج لمبادلة الإحداثيات العقدية المتوسطة وشكل آخر لا يبادل البيانات. بالإضافة إلى ذلك ، تم تنفيذ تمهيد لابلاسيان القياسي لإثبات فعالية النموذج المقدم. وقد اظهرت النتائج أنه عند دقة واحدة ومزدوجة ، فإن تطبيقات وحدات معالج الرسومات المطورة باستخدام هياكل البيانات التي يمثلها تخطيط صفيف الهياكل المتوازية تحقق أفضل كفاءة. ويتضح أيضا أن النموذج الذي يحتاج لمبادلة الإحداثيات العقدية المتوسطة هو دائما أبطأ من تلك التي من دون مبادلة للبيانات.

1 Introduction

General purpose computing on modern Graphics Processor Units (GPGPU) can significantly reduce computational cost by performing massively parallel computing. The GPU architecture is ideally suitable for data-parallel computations with high arithmetic intensity such as numerical computations and related applications. For example, the benefits of GPGPU have been exploited in computational fluid dynamics [1,2], mesh generation [3,4], and mesh optimization [5,6] by using

GPU programming models such as CUDA [7] and OpenCL [8].

In numerical analysis such as FEM, the quality of computational meshes plays a key role on the accuracy and efficiency of final results [9–11]. Therefore, numerous mesh generation approaches have been proposed to obtain high quality meshes; see a survey in [12]. In addition, original meshes are needed to be optimized after being created to improve the mesh quality. A variety of methods have been developed to deal with this issue. Mesh smoothing is one of the most commonly used strategies for improving the mesh quality by adjusting the positions of mesh vertices/nodes without altering the topology of the mesh. The basic idea behind mesh smoothing is to make a mesh achieve its highest quality by repositioning all vertices in the mesh.

Mesh smoothing for large meshes are generally computational expensive. In order to reduce the computational time cost, several efforts have been carried out to implement mesh smoothing in parallel on shared-/distributed- memory computers. For example, Freitag et al. [13] designed a parallel algorithm for smoothing independent sets of vertices simultaneously on a parallel random access machine (PRAM) model. Jiao et al. [14] developed a parallel feature preserving mesh smoothing algorithm for surface meshes on distributed memory computers with up to 128 processors. Gorman et al. [15] presented an optimization based mesh smoothing algorithm for anisotropic mesh adaptivity by using hybrid OpenMP/MPI programming methods and graph coloring. Bentez et al. [16] proposed a new parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes by taking advantage of OpenMP on many-core shared memory computers. Most recently, Sastry and Shontz [17] developed a parallel log barrier mesh untangling and mesh quality improvement algorithm for distributed-memory machines using OpenMPI 2.0.

Most of the above implementations of mesh smoothing are conducted using the parallel programming models OpenMP and/or MPI. These parallel implementations are fast and effective; but an obvious problem is that high performance computers or clusters are needed to develop such implementations, which leads to high resource cost. An alternative and practical solution is to develop the parallel implementations on modern GPUs since massively parallel computing performed on GPUs requires relatively low cost.

D'Amato and Lotito [18] proposed an efficient and parallelizable method to perform mesh optimizations including mesh smoothing and topological changes for surface triangular meshes on the GPU. In combination with matrix-based multigrid methods, Heuveline et al. [19] developed parallel smoothers for smoothing unstructured, locally refined meshes using multicore CPUs and GPUs. Based on a GPU mesh smoothing filter, Monch et al. [20] designed an OpenGL-based mesh smoothing filter for surface mesh models derived from medical image data.

In order to guarantee that there are no inverted elements created in mesh smoothing and mesh quality is constantly improved, D'Amato and Vnere [5] proposed an efficient but expensive strategy for smoothing tetrahedral meshes: when moving a node to improve the quality of a set of local elements, several "candidate positions" are first created according to the distances to the original location; and then the best position will be selected by comparing the quality of local elements.

The studies described above are mostly problem-specific and based on vertices' movement. Among the mesh smoothing approaches that are computationally iterative and vertex-based, the most popular and simplest method is Laplacian smoothing, which in its original form [21] moves each node to be at the average of its incident neighboring nodes.

A key design issue for generating efficient GPU code is the form in which data should be organized. There are generally two major choices of the data layout: the array-of-structures (AoS) and the structure-of-arrays (SoA); see Fig. 3. Organizing data in AoS layout leads to coalescing issues as the data are interleaved, while the organizing of data according to the SoA layout can typically make full use of the memory bandwidth since there is no data interleaving. Furthermore, global memory accesses are always coalesced when using the SoA layout.

In this paper, specifically aiming at the Laplacian-based mesh smoothing, a generic paradigm for utilizing the power of a single NVIDIA GPU is developed to improve the computational efficiency by using the GPU programming model CUDA [7]. The data layouts such as AoS and SoA for representing mesh data structures on the GPU are first discussed; and then a practical solution to dealing with the data dependencies in the iterative mesh smoothing process is introduced. Two forms of iteration in mesh smoothing also analyzed by comparing the performances when selecting different intermediate data (i.e., the coordinates of all vertices in previous pass of iteration, or alternately in the current pass) for calculating the smoothed vertices' positions. In addition, the standard Laplacian smoothing is implemented to demonstrate the effectiveness of the proposed paradigm.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to the Laplacian mesh smoothing. Section 3 introduces the generic paradigm for accelerating Laplacian-based mesh smoothing on the GPU. Section 4 concentrates mainly on the presented GPU implementations of the standard Laplacian smoothing using three sets of mesh data structures represented by different data layouts. Section 5 presents several experimental tests and discusses the results. Finally, Sect. 6 draws some conclusions.

2 Laplacian Mesh Smoothing

Mesh smoothing is one of the most popular approaches that are capable of improving mesh quality by repositioning nodes /vertices in meshes while keeping the mesh topology unchanged. One of the widely used mesh smoothing method is the Laplacian smoothing [21], which relocates each node in a mesh to the geometric center of its neighboring nodes. Laplacian smoothing is straightforward and easy to implement but does not always guarantee improvement in mesh quality. In fact, it is possible to create inverted or even invalid elements in concave regions. Hence, various variations such as the *weighted Laplacian smoothing* [22,23] and *constrained / smart Laplacian smoothing* [24,25] have been developed to improve the performance of the original form of Laplacian smoothing.

2.1 Commonly-Used Variations

2.1.1 Standard Laplacian Smoothing

In the standard form of Laplacian smoothing [21], a newly smoothed position of a node in a mesh is directly the geometric center of its neighbors (i.e., incident nodes). More formally, the smoothing operation can be simply described as follows:

$$\overline{x_i} = \frac{1}{N}\sum_{j=1}^{N} x_j, \tag{1}$$

where N is the number of neighboring nodes to node i, and $\overline{x_i}$ is the new position for node i.

2.1.2 Weighted Laplacian Smoothing

An improved version of the standard Laplacian smoothing is the weighted variation [22], where the weights are calculated according to some kinds of factors such as edge-length or face-area. For example, the length-weighted Laplacian smoothing calculates the weights according to the length of edges that locate around the mesh node being smoothed. Therefore, this variation is highly sensitive to element edge lengths and attempts to average these lengths to form better shaped elements. However, similar to the standard version, the length-weighted Laplacian smoothing still has difficulties when smoothing elements in the concave regions. The smoothing operation of the weighted Laplacian smoothing can be simply described as follows:

$$\overline{x_i} = \sum_{j=1}^{N} \omega_j \cdot x_j, \quad \sum_{j=1}^{N} \omega_j = 1, \tag{2}$$

where N is the number of neighboring nodes to node i, and $\overline{x_i}$ is the new position for node i, ω_j is the weight for the neighboring node j. Obviously, The standard Laplacian smoothing is the special and simpler form of weighted version, in which all the weights for a vertex are equivalent (i.e., $\omega_j = 1/N$).

2.1.3 Smart Laplacian Smoothing

As mentioned above, inverted or even invalid elements are possibly created in highly concave regions when using Laplacian smoothing. This negative behavior is certainly needed to be avoided. An effective solution to the above problem is to check whether there is an improvement in mesh quality when repositioning a node. This solution is ”smart” to determine whether or not to relocate a node; and the variation in this case is referred to as smart Laplacian smoothing [24].

The basic idea behind smart Laplacian smoothing is simple. For a node being smoothed, a patch of elements sharing this node are first recorded and accessed the mesh quality. Then a newly smoothed position of this node is calculated according to the Laplacian smoothing operations such as the ones described in Eqs. (1) and (2). The quality of the patch is assessed again using the new nodal position. If there is an improvement in the quality of the elements in the patch, the new nodal position will be accepted; otherwise, the node will not be relocated. In short, smart Laplacian smoothing relocates a node at a new position only when the mesh quality is improved.

2.2 Iteration Forms

When calculating the smoothed coordinates of vertices according to the smoothing operations, there are typically two forms in terms of selecting the coordinates of neighboring nodes. More specifically, in one pass of iteratively calculating all smoothed nodal positions, the coordinates of neighboring nodes to a smoothed node can be derived from: (1) the old coordinates calculated in previous pass of iterating, and (2) the new coordinates that have been calculated in current pass of iterating. For example, when calculating the smoothed coordinates $\overline{x_i}$ of the node i in the iteration pass $(q + 1)$, the coordinates of the neighboring nodes to the node i can be completely derived from the previous iteration pass q or partially derived from both the previous iteration pass q and the current iteration pass $(q + 1)$. The above two forms of the standard Laplacian smoothing can be simply illustrated using the following formulations.

Form A:

$$\overline{x_i^{q+1}} = \frac{1}{N}\sum_{j=1}^{N} x_j^q, \tag{3}$$

where N is the number of neighboring nodes to node i and $\overline{x_i^{q+1}}$ is the new position for node i in the iteration pass $(q+1)$.

Form B:

$$\overline{x_i^{q+1}} = \frac{1}{N}\left(\sum_{j=1}^{N_q} x_j^q + \sum_{k=1}^{N_{q+1}} x_k^{q+1}\right), \quad \begin{cases} 0 \leqslant N_q \leqslant N \\ 0 \leqslant N_{q+1} \leqslant N \\ N_q + N_{q+1} = N \end{cases}, \tag{4}$$

where N is the number of neighboring nodes to node i and $\overline{x_i^{q+1}}$ is the new position for node i in the iteration pass $(q+1)$. N_q and N_{q+1} are numbers of neighboring nodes derived from the iteration passes q and (q+1), respectively. Obviously, the Form A is a special case of the Form B where $N_{q+1} = 0$.

3 The Paradigm

3.1 The Working Process

Laplacian smoothing is an iterative process, where in each step every vertex of the mesh is moved to the barycenter of its neighbors. In this context, boundary vertices of the mesh are constrained not to be moved, but internal vertices are simultaneously moved to the averaging center of its neighboring vertices. And then the process is iterated a number of times.

According to the working process of Laplacian mesh smoothing, the entire procedure can be divided into four sub-procedures. Each sub-procedure is responsible for performing a task in the process of Laplacian smoothing that can be parallelized on the GPU; and each sub-procedure is then mapped to a CUDA kernel; see Fig. 1. The ideas and considerations behind those four sub-procedures will be described in more details.

3.1.1 Finding Neighbors

Typically, in Laplacian mesh smoothing, the neighbors (i.e., adjacent nodes) of each mesh node need to be found before iteratively calculating the smoothed nodal location. A Brute-Force method is to loop over all elements and check which elements contain a specific node; and thus it needs to loop n times, where n is the number of vertices.

An improved and practical approach is to find the neighbors according to the topology of elements. For example, for a triangle, both the second and the third vertices are the neighbors of the first vertices; for a quadrilateral, the second and the fourth vertices are the neighbors of the first vertex. In this approach, only one time of looping over all elements is needed. The neighbors of a vertex in an elements can be directly determined according to the connectivity of the element.

The above method is well suited to be implemented in sequential programming pattern on CPU. When programming on the GPU, a potential useful consideration is that whether it is possible to determine the neighbors for all vertices in parallel. For example, if there are m elements, then m threads will be allocated; and each thread is responsible for finding the neighbors of each vertex within only one element.

This idea seems to be effective. However, a problem arises due to the data conflict. For a mesh node, a private array is usually allocated to store the indices of its neighboring nodes. When only one thread finds one of the node's neighbors and then writes the index of the neighbor into the private array for storing, there is only one operation of writing in this case, and thus there is no data conflict. Unfortunately, the above safe condition cannot be guaranteed in parallel pattern: there are probably more than two threads finding and then writing the indices of the neighboring nodes into the same element of the same array (i.e., the same memory address) simultaneously. Hence, data conflict arises and eventually leads to failures.

In order to deal with the above problem, the practical approach that is generally implemented in sequential programming pattern is still adopted. However, it is implemented on the GPU rather than on the CPU; and only one thread and one thread block is allocated to determine the neighbors of all vertices. Obviously, this solution is inefficient but effective.

3.1.2 Determining Constrained Vertices

In all types of mesh smoothing, constrained vertices are those nodes that cannot be moved completely freely. Those vertices (1) cannot be moved or (2) can only be relocated while meeting some constraints such as lying on a plane. These constrained vertices are used to preserve the features such as shape and volume of mesh models being smoothed. Typically, constrained vertices are derived from (1) boundary of mesh models and (2) user-specified constraints such as fixed edges or faces. More sources of constrained vertices are usually needed to be determined according to the type of meshes (e.g., planar mesh, surface mesh, or volume mesh). In this paper, only the Laplacian-based mesh smoothing

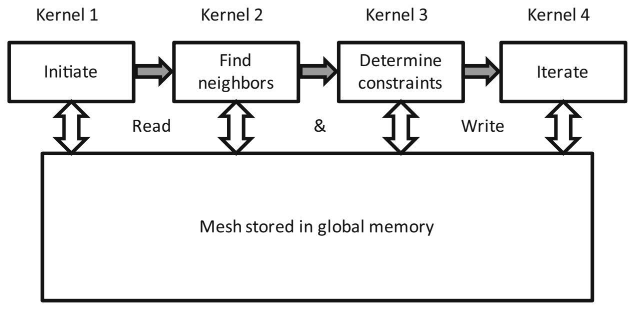


Fig. 1 The working process

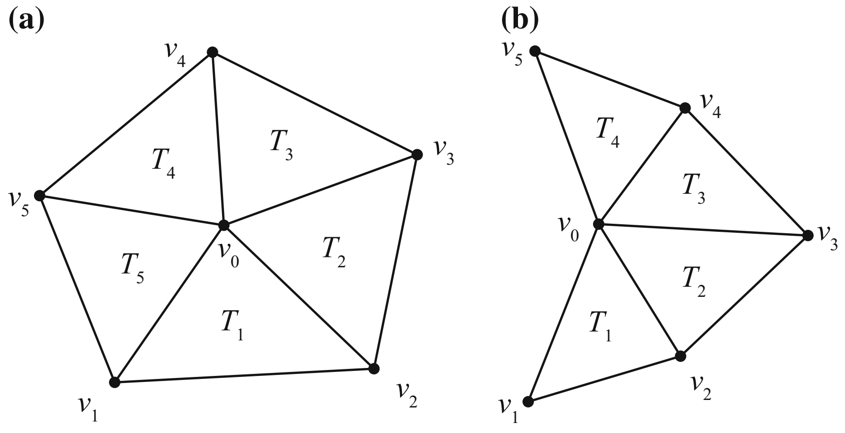


Fig. 2 Determining boundary vertices according to neighbors

approaches applied for planar meshes are discussed. Thus, the constrained vertices are only the boundary nodes.

Before determining the boundary vertices, the neighboring nodes of each vertex have been found in previous step. An efficient method for checking whether a node is a boundary one is to take advantage of its neighbors. For example, for the node v_0 in Fig. 2a, its neighbors are the nodes $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\{v_3, v_4\}$, $\{v_4, v_5\}$, and $\{v_5, v_1\}$. Each of its neighbors are recorded twice due to calculating from two adjacent triangles. In other words, any edge containing a internal vertex would be shared by two adjacent triangles. Therefore, the basic idea behind determining the boundary vertices is straightforward: if all the neighbors of a vertex, e.g., the node v_0 in Fig. 2a, have been recorded twice, then the vertex is internal; otherwise, it is a boundary vertex (e.g., the node v_0 in Fig. 2b).

Obviously, the above approach for determining boundary vertices can be easily implemented in parallel on the GPU: if there are n vertices in a mesh, then n threads need to be allocated; and each thread is responsible for checking a vertex whether it is boundary vertex or not. The indices of neighboring nodes of each vertex are stored in an array that is owned by the target vertex, and thus can be accessed without having conflicts with that of other vertices.

3.1.3 Iterating

Laplacian smoothing is computationally iterative. The iterative process of calculating smoothed nodal positions can be very easily implemented in sequential programming pattern. However, when programming parallel computations on the GPU, the procedure of iterating in Laplacian smoothing is not easy to implement. Several important issues need to be carefully handled.

The first issue that needs to be considered is the data dependency. In each pass of iteration, the new smoothed nodal positions are calculated according to the old nodal coordinates obtained in previous pass of iteration. Thus, in the parallel implementation of Laplacian smoothing on the GPU, the next pass of iteration can only be performed when all the threads have finished their calculating of smoothed positions in the previous pass of iteration.

However, the above condition cannot be guaranteed in practice. This is due to the inherent features of current GPU programming model: in CUDA, a set of threads are grouped into thread blocks; all threads within the same thread block can be synchronized; but the synchronization between different thread blocks does not exist. If threads are grouped into more than one block, then the computations carried out by all threads cannot be guaranteed to be finished at a user-specified moment. In short, there is no barrier of synchronization between thread blocks.

A practical solution to above problem is to map the calculating of smoothed nodal positions into only one block. More specifically, each thread within the only one block is responsible for calculating the smoothed positions of several nodes; and thus the calculation of new positions for all vertices can be synchronized in one iteration step. For example, assuming there are 2,048 vertices being smoothed, each thread is responsible for calculating the smoothed locations of 4 vertices when there are 512 threads allocated in the thread block. Before calculating the next pass of smoothed positions of the total 2,048 vertices, current pass of calculation can be guaranteed to be completely finished using the barrier `__syncthreads()`.

3.2 Key Issues in Design

3.2.1 Data Layout in Memory

A key design issue for generating efficient GPU code is the form in which data should be organized when operating on multi-valued data such as sets of points or pixels. In general, there are two major choices of the data layout: the AoS and the SoA; see Fig. 3.

(a)

```
struct Pt {
  float x;
  float y;
};
struct Pt myPts[N];
```

(b)

```
struct Pt {
  float x[N];
  float y[N];
};
struct Pt myPts;
```

Fig. 3 Data layouts: **a** AoS and **b** SoA

Organizing data in AoS layout leads to coalescing issues as the data are interleaved. For example, without forcing to be aligned (e.g., having not to use the alignment specifier `__align__()` in CUDA), performing an operation on a set of 2D points illustrated in Fig. 3a that only requires the variable x will result in a 50 % loss of bandwidth and waste of L2 cache memory.

In contrast, the organizing of data according to the SoA layout (see Fig. 3b) can typically make full use of the memory bandwidth since there is no data interleaving. In addition, global memory accesses are always coalesced when using this type of data layout; and usually high global memory performance can be achieved.

The SoA data layout is beneficial in many application cases. Farber [26] suggested that from a GPU performance perspective, it is preferable to use the SoA layout. This argument was demonstrated by the example of sorting SoA and AoS structures with *Thrust*; it was reported that a 5-times speedup can be achieved by using a SoA data structure over a AoS data structure [27].

Similarly, in order to evaluate the effective performance of the two representations, i.e., the SoA and AoS layouts, on the GPU, Govender et al. [28] ran a simulation of 2 million particles using their discrete element simulation framework BLAZE-DEM, and found that AoS is three times slower than SoA. An opposite argument was presented in [2]. In the library framework for the solution of unstructured mesh applications, OP2, Giles et al. [2] and Mudalige et al. [29] preferred to use the AoS layout to store mesh data for better memory accesses performance.

And in order to make the data structures represented by different data layouts flexible in different applications, several efforts have been carried out to transform different layouts (e.g., AoS and SoA) to each other [30–33].

The above mentioned applications show that memory access patterns (e.g., AoS and SoA) are critical for performance, especially on parallel architectures such as GPUs. However, it is not always obvious which data layout will achieve better performance in a particular application. In this work, both of the above two data layouts are tested. First, the mesh data structures represented by both the AoS and SoA data layouts are designed; and then the standard Laplacian smoothing is implemented using the designed mesh data structures. Details are described in Sect. 4.

3.2.2 Data Dependencies

A data dependency is a situation in which some calculation in a program requires the result of a previous calculation. Data dependencies are typically seen in two forms: (1) one calculation is dependent on one or more calculations around it; and (2) there are multiple passes over a dataset and there exists a dependency from one pass to the next. For example, a loop that has results from one iteration feeding back into a future iteration of the same loop is considered to have a data dependency.

In this work, the research interest is mainly focused on the second form, i.e., the data dependencies existing in different passes of iterating. Laplacian smoothing is an iterative procedure. The iterative calculations of the smoothed positions for all mesh vertices are repeated until converging. In each pass of iterating, new positions are calculated according to the smoothed positions calculated in previous pass. Thus, there exists a data dependency from one pass to the next.

This data dependency issue causes a problem of synchronizing. The next pass of all smoothed positions cannot be calculated until the previous pass of calculations has been finished. Therefore, a barrier of synchronization is needed to guarantee the calculations of all smoothed positions in one iteration step having been finished before calculating the next pass of smoothed positions.

However, in CUDA, there is no global barrier of synchronization for all threads. The computations carried out by all threads within a single kernel can only be guaranteed to be completely finished after invoking this kernel. A practical solution to the problem of synchronizing is to map the iterative calculations of all smoothed positions into only one thread block. Each thread within the thread block is responsible for calculating the smoothed positions of several rather than one vertex. And the barrier `__syncthreads()` is used to guarantee all threads within the only one block finishing calculating one pass of all smoothed positions. This solution has been described in Sect. 3.1.3.

Both the Form A and the Form B of Laplacian smoothing (see Sect. 2.1.1) have data dependencies between two passes of iteratively calculating the smoothed nodal positions. This type of data dependency arises between a previous pass and a next pass, which therefore can be referred to as *inter-pass* data dependency.

Different from the Form A, the Form B has another type of data dependency. As described in Sect. 2.1.1, when calculating the smoothed position of the node i in the Form B [see Eq. (4)], the coordinates of neighboring nodes are partially derived from the old coordinates calculated in the previous pass and the new coordinates calculated in the current pass. This means some of the neighboring nodes to the node i have been smoothed in the current pass of iterating, and then their newly smoothed coordinates are adopted to calculate the smoothed position of the node i. This also means the calculating of the smoothed position of the node i also depends on which neighbors having been smoothed in the current pass of iterating. This type of data dependency can be referred to as *in-pass* data dependency, which differs from the *inter-pass* data dependency.

In the Form B, a problem may arise due to the in-pass data dependency issue: assuming a thread t_0 is calculating the smoothed position of the node i and another thread t_1 is simultaneously calculating the coordinates of one of the neighbors to the node i, e.g., the node j, thus the memory storing the coordinates of the node j is being accessed concurrently by both the threads t_0 and t_1.

If and only if both the threads t_0 and t_1 are scheduled within the same *warp* (or *half-warp* on some devices), a data race condition appears due to concurrent multiple accesses to the same memory. And in this data race condition, the multiple accesses are no longer executed in parallel but in sequence. In addition, the order of thread executions is undefined. In other words, the thread t_0 may first read the memory storing the coordinates of the node j and then the thread t_1 writes the newly smoothed coordinates of the node j into the memory; or the thread t_1 first writes the new coordinates into the memory and then accessed by the thread t_0. These two cases cannot be specifically determined in practical executions.

However, no matter which of the above two cases being performed in practice, the implementation of the Form B will run correctly. In the first case, the old coordinates of the node j is used to calculate the smoothed position of the node i; and in the second case, the new coordinates of the node j is used. The differences between the two cases can be simply thought as follows: in the first case, the number of the neighbors selected from previous pass of iterating, i.e., N_q, increases with 1 and correspondingly the number of neighbors chosen from current pass of iterating, i.e., N_{q+1}, decreases with 1; see the Eq. (4). In the second case, the opposite situation occurs. Therefore, both of the above two cases are the specific situations of the Form B; and the solution to the problem caused by in-pass data dependency is not needed.

4 GPU Implementations of Standard Laplacian Smoothing

This section will first introduce several groups of mesh data structures represented by three types of data layouts, i.e., the mis-aligned array-of-structures (Mis), the aligned AoS, and the SoA. And then, some details of four CUDA kernels in the GPU implementations of Laplacian smoothing will be described. These GPU implementations are developed on both single and double precision.

4.1 Mesh Data Structures

Before introducing the data structures for representing planar triangular meshes on the GPU, the data structures on the CPU are first described; see the Listing 1. Those three structures `CVert`, `CTrgl`, and `CMesh` are used to represent a vertex, a triangle, and a triangular mesh, respectively. The meanings of the components in each structure are also explained. Noticeably, this group of data structures is quite similar to the group represented by the layout Mis (Listing 2).

4.1.1 Meshes Represented by the Layout Mis

The data structures of planar triangular mesh are represented using the Mis layout is illustrated in the Listing 2. Those three structures `cuVert_MIS`, `cuTrgl_MIS`, and `cuMesh_MIS` are used to represent a vertex, a triangle, and a triangular mesh, respectively. The meanings of the components in each structure are also explained. Obviously, the accesses to global memory when using these data structures are non-coalesced.

4.1.2 Meshes Represented by the Layout AoS

In CUDA, global memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads [7].

To maximize global memory throughput, it is therefore important to maximize coalescing by using data types that meet the size and alignment requirement. Any access to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned. If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing [7].

The alignment requirement is automatically fulfilled for the built-in types like `float2` or `float4`. For structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers `__align__(8)` or `__align__(16)`. If structure size exceeds 16 bytes, it can't be efficiently read or written, since more than one global memory non-coalescable load/store instructions will be generated, even if `__align__` option is supplied.

Therefore, the data structures represented by AoS layout on single precision are first redesigned; then the spefiers are added into the structures `cuVert_AOS` and `cuTrgl_AOS`; see the Listing 3. Different from the structure `cuVert_MIS` listed in the Listing 2, the component `neig[64]` is moved into a separate structure to let the size of `cuVert_AOS` be 16 bytes.

Similarly, for the data structures represented by AoS layout on double precision, several new data structures with less components (e.g., `cuCoor_AOS` and `cuInfo_AOS`) are created to meet the size and alignment requirements; see the Listing 4.

4.1.3 Meshes Represented by the Layout SoA

The data structures of planar triangular mesh are represented using the SoA layout is listed in the Listing 5. The structures `cuVert_SOA`, `cuTrgl_SOA`, and `cuMesh_SOA` are used to represent a vertex, a triangle, and a triangular mesh, respectively. The meanings of the components in each structure are also explained. When using these data structures, the global memory accesses are naturally coalesced.

4.2 Implementation Details

4.2.1 The Kernel for Initiating

The first kernel is invoked to initiate the properties of each mesh vertex, more specifically, to set (1) all vertices as internal rather than boundary vertices and (2) the number of neighbors to each vertex as zero. The implementation of this kernel is quite straightforward; and each thread takes responsibilities for initiating the properties of one vertex.

4.2.2 The Kernel for Finding Neighbors

The second kernel is responsible for finding the neighbors (i.e., adjacent nodes) to each vertex in the planar triangular mesh. Due to the data conflict issue explained in the Sect. 3.1.1, this procedure of finding neighbors is as the same as the sequential version implemented on CPU. Thus, only one thread and one thread block are needed to be allocated.

4.2.3 The Kernel for Determining Constrained Vertices

The third kernel is called to determine which vertices are boundary ones by checking the indices of neighbors. The idea behind this procedure is quite simple; see more descriptions in Sect. 3.1.2. The indices of the neighbors to a vertex is stored in an array `neig[64]`. Boundary vertex can simply be determined by checking whether all neighbors to a vertex are stored twice in the array `neig[64]`. Each thread is invoked to check whether a vertex is boundary vertex or not.

Noticeably, the maximum number of the neighbors to any vertex is limited to 32 (i.e., 64/2). Theoretically, both the number of neighbors and the size of the array `neig[]` can be arbitrary. However, in practical applications, the number of neighbors and the number of the triangles shared a specific vertex cannot be too large. This is because when there are many triangles, e.g., 32, share the same vertex, then the maximum minimum angle of all the 32 triangles is $360^\circ/32 = 11.25^\circ$. And in this case, the quality of mesh is bad due to the sliver/sharp element. In practical applications such as numerical analysis, meshes with sliver elements are not recommended and need to be avoided.

In the presented implementations, the size of the array `neig[]` is specifically set to 64; and thus the maximum number of the neighbors to one vertex is 32. Failures will arise when the above limit is exceeded.

4.2.4 The Kernel for Iterating

The final and key kernel is responsible for iteratively calculating the smoothed positions of all vertices. Due to the data dependencies existing in different passes of iterative calculations, only one thread block is allocated. Each thread within the thread block is invoked to calculate the smoothed positions of several vertices in one pass of iteration. The barrier of synchronization `__syncthreads()` is used to guarantee all threads within the only one block finishing calculating one pass of all smoothed positions.

In addition, a *while* loop is used to make the iterative calculations repeated until converging. The criteria for terminating the iterating is that the maximum differences between the new and old nodal coordinates generated in two passes of iterating is less than a user-specified *threshold* (this tolerance is set to 10e−6). Firstly, the local maximum difference is sequentially found within one thread; and then parallel reduction is carried out to find the global maximum difference of all vertices. If the global maximum difference is less than the threshold, then the iteration terminates.

Noticeably, the execution of this kernel exists only when all threads meet the termination criteria. Thus, after obtaining the maximum difference in a single thread, the maximum differences is then broadcasted into all threads; and in the next pass of iterating, each thread will make a decision whether or not to terminate the iterating.

5 Results and Discussion

5.1 Results

The GPU implementations are evaluated using the NVIDIA GeForce GT640 (GDDR5) graphics card with 1GB memory and CUDA v5.5. Note that the GeForce GT640 card with memory GDDR5 has the Compute Capability (CC) 3.5, while it only has Compute Capability 2.1 with the memory DDR3. The CPU experiments are performed on Windows 7 SP1 with an Intel i5-3470 CPU (3.2 GHz and 4 Cores) and 8GB of RAM memory. For each set of the testing data, one CPU implementation and three GPU implementations are carried out on both single precision and double precision.

The original unsmoothed triangular meshes are generated according to the standard Delaunay triangulation algorithm. First, five sets of uniformly distributed points in 2D are randomly created using the generator provided by Qi et al. [3]; and then Delaunay meshes are generated for these sets of discrete points using the library `Triangle` [34].

Five triangular meshes are created, which are composed of 1, 5, 10, 50, and 100 K vertices, respectively. The mesh illustrated in Fig. 4a is one of the testing triangular meshes that consists of 1,000 vertices and 1,977 triangles; and the mesh presented in Fig. 4b is the corresponding smoothed result using the standard Laplacian smoothing.

5.1.1 Single Precision

The experimental results on single precision are presented in Figs. 5 and 6. According to those benchmark results, the GPU version implemented using the mesh data structures represented by aligned AoS achieves the best efficiency; and the second best is the version developed using the data structures represented by the layout SoA. The worse GPU version is the one implemented based on the mis-aligned array-of-structures (Mis). For the best GPU version (i.e., the GPU–AoS version), the Form A achieves the speedups of about 13×–21× over the corresponding CPU version, while the speedups are about 7×–19× for the Form B; see Fig. 6.

5.1.2 Double Precision

The benchmark results on double precision are shown in Figs. 7 and 8. Similar to the performance on single precision, among the three GPU versions, the one implemented using the mesh data structures represented by AoS is still the most efficient. However, the speedups achieved by this version is about 11×–17× and 8×–16× for the Form A and Form B, respectively; see Fig. 8. The performance of this GPU version over the CPU version on double precision is slightly worse than that on single precision.

5.2 Discussion

Mesh smoothing is a well established technique to improve the quality of computational meshes in numerical analysis and related fields. When smoothing large meshes that consist of large numbers of nodes and elements, the computational cost is usually too high. In order to improve the computational efficiency, several efforts have been carried out to implement the mesh smoothing approaches on the GPU [5,18–20]. However, these parallel implementations are mostly problem-specific or integrated with other types

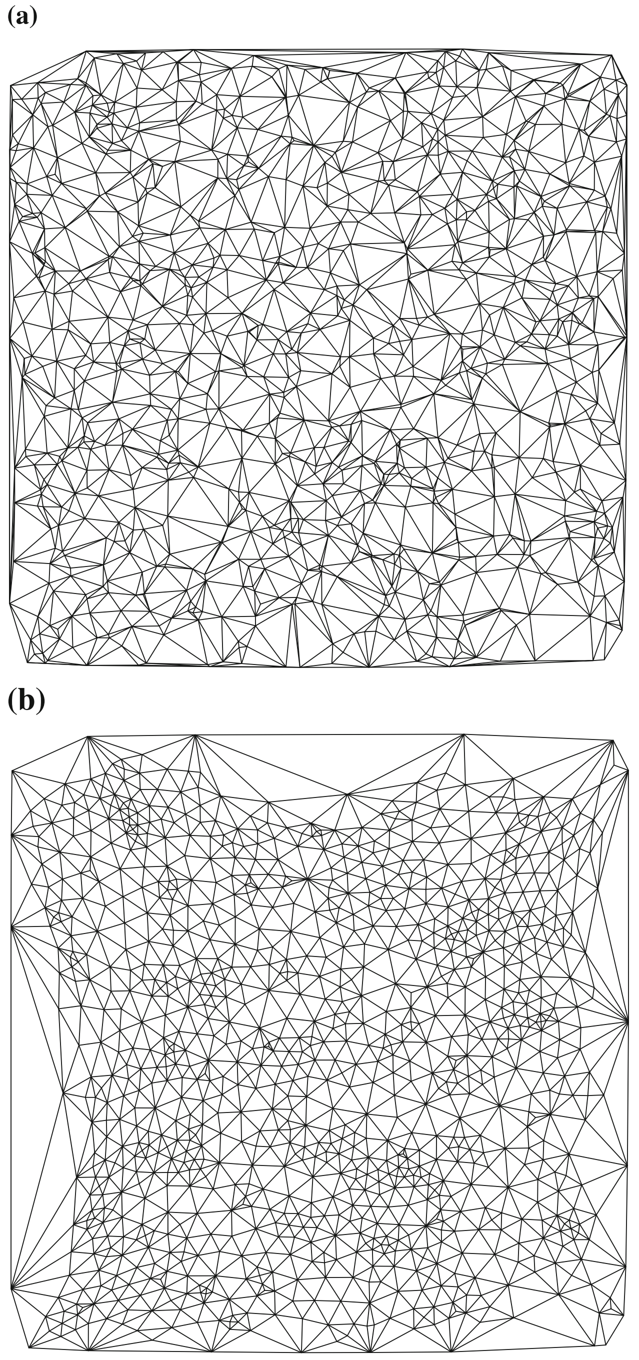


Fig. 4 An application example of Laplacian smoothing. **a** Original mesh, **b** Smoothed mesh

of mesh optimizations. To the best of the authors' knowledge, there are no existing studies having been performed to introduce a generic paradigm specifically for the Laplacian mesh smoothing.

In this paper, a generic paradigm for accelerating the Laplacian-based mesh smoothing is developed by exploiting a single GPU. In order to demonstrate the working process of the paradigm, two forms of the standard Laplacian smooth-

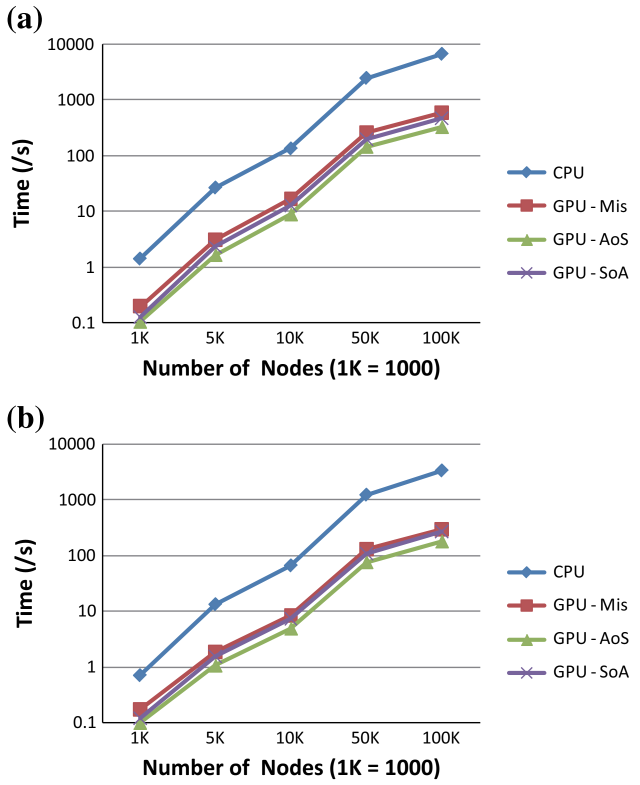


Fig. 5 Execution time on single precision. **a** Form A of LS, **b** form B of LS

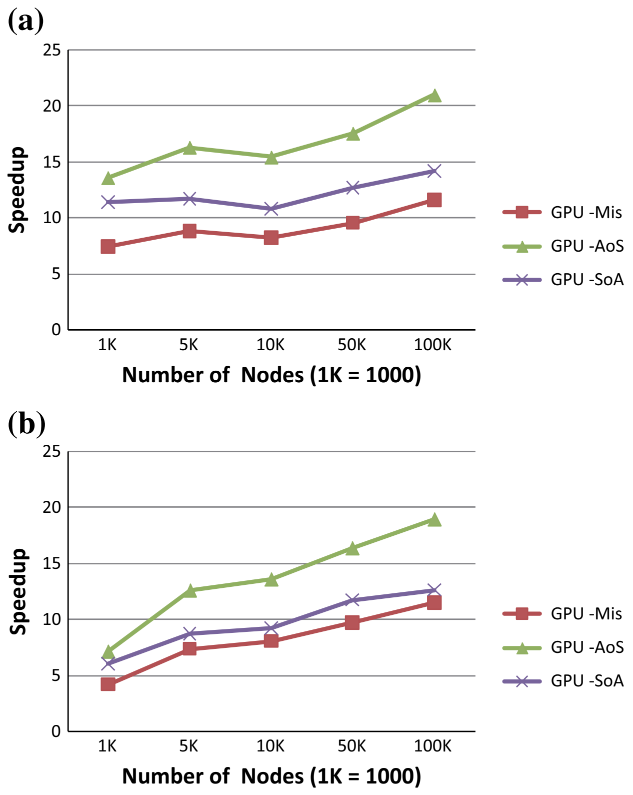


Fig. 6 Speedups of GPU implementations over CPU implementation on single precision. **a** Form A of LS, **b** form B of LS

ing are implemented using different mesh data structures on both single and double precision. The performance of these GPU implementations are analysed as follows.

5.2.1 Efficiency of Each Sub-Procedure (Kernel)

The entire process of Laplacian mesh smoothing is divided into four sub-procedures. Each sub-procedure is responsible for performing a task in the process of Laplacian smoothing, which is mapped to a CUDA kernel; see Fig. 1. This section will investigate the efficiency of those sub-procedures. For the implementations of the standard Laplacian smoothing developed using the data structures represented by the AoS layout, the percentage of the running time for each sub-procedure (i.e., a CUDA kernel) is tested on single and double precision; see the results listed in Tables 1, 2, 3, 4.

According to the efficiency performance of each sub-procedure, it can be observed that: the sub-procedure of iterating (i.e., the Kernel 4 in Tables 1, 2, 3, 4) is the most computational expensive step, while the other three sub-procedures generally cost $<1\,\%$ time in most cases. Among the first three sub-procedures, the second sub-procedure typically needs much more execution time than that of the rest two. This is perhaps due to the cause that only one thread and one thread block are allocated to determine the neighbors of all vertices. In this sub-procedure, the power of massively parallel computing on the GPU is obviously not fully exploited. However, the cost of the second sub-procedure is quite small when compared to that of the fourth one. Thus, the solution of allocating only one thread and one block is practical. In addition, more than 98 % time is spent on the fourth sub-procedure in almost all cases. Therefore, future optimizations need to focus on this time-consuming sub-procedure.

5.2.2 Performance Impact of Data Layouts

On both single and double precision, the GPU implementations developed using the data structures represented by AoS layout achieve the best efficiency. And in the two groups of GPU implementations developed using the data structures represented by layouts SoA and Mis, the performance of the first group is better than the second.

The best performance obtained by the GPU version based upon the AoS data structures is due to the aligned global memory accesses. The performance differences between with and without aligning the accesses to global memory can be clearly observed in Figs. 6 and 8. Noticeably, the GPU version implemented using the SoA data structures is better than that implemented using the Mis data structures; this behavior is mainly due to the coalesced memory accesses.

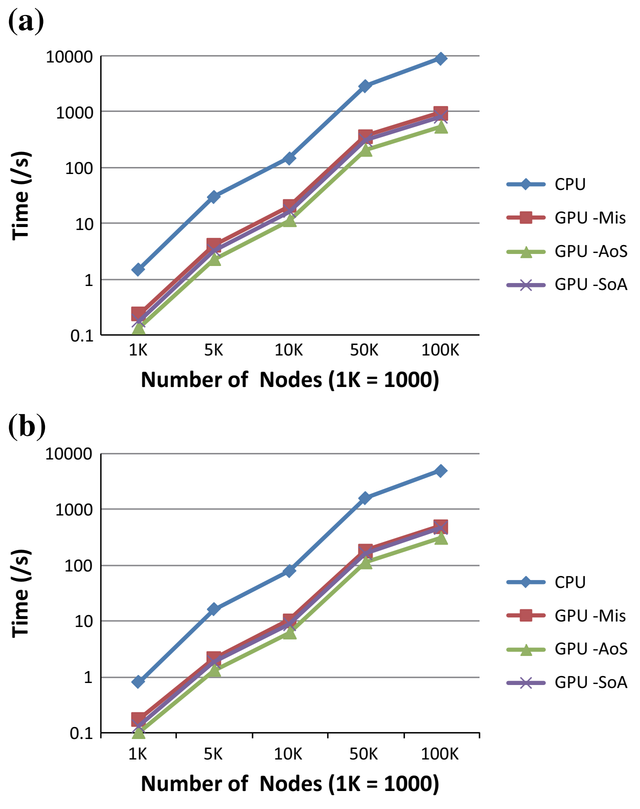


Fig. 7 Execution time on double precision. **a** Form A of LS, **b** form B of LS

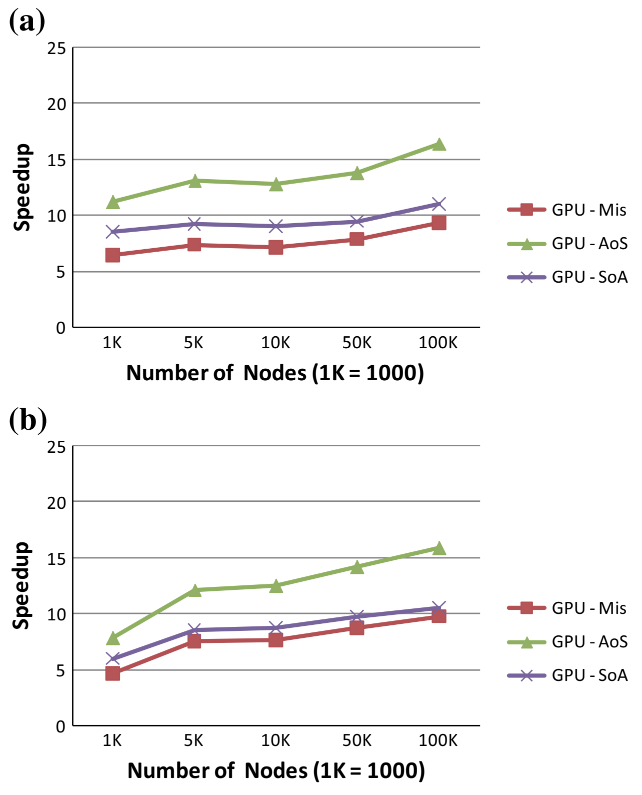


Fig. 8 Speedups of GPU implementations over CPU implementation on double precision. **a** Form A of LS, **b** form B of LS

According to above testing results, it is recommended that the first choice of data layouts for representing mesh data structures in Laplacian smoothing is the layout aligned AoS. However, when the size and alignment requirement cannot be met in some cases, e.g., when representing a vertex in 3D on double precision (i.e., when using the structure `struct vertex {double x, y, z;};`), the best option is the layout SoA.

5.2.3 Performance of the Form A and the Form B

For the two forms of Laplacian mesh smoothing, the Form A that needs to swap intermediate nodal coordinates is always slower than the Form B that does not swap data. This behavior can be obviously observed for all CPU and GPU implementations on both single and double precision. The Form B achieves the speedups of about 1.5× –2.0× over the Form A; see Fig. 9.

The above results are caused by more calculations due to swapping intermediate nodal coordinates during iterating. Furthermore, for each GPU implementation of the Form A, an array residing in global memory is needed to be allocated to store and swap intermediate nodal positions. Thus, the global memory accesses in the implementations of the Form A is much more that those of the Form B.

Another cause is that the convergence speed of the Form A is much lower than that of the Form B; and thus the

Table 1 Percentage of running time for each kernel of the Form A on single precision

Size	Kernel 1	Kernel 2	Kernel 3	Kernel 4
1 K	0.842	2.339	0.655	95.510
5 K	0.084	0.751	0.096	99.032
10 K	0.027	0.287	0.029	99.645
50 K	0.005	0.093	0.008	99.891
100 K	0.005	0.083	0.007	99.904

Table 2 Percentage of running time for each kernel of the Form B on single precision

Size	Kernel 1	Kernel 2	Kernel 3	Kernel 4
1 K	0.866	2.406	0.674	95.669
5 K	0.125	1.115	0.143	98.572
10 K	0.047	0.499	0.051	99.393
50 K	0.010	0.173	0.015	99.801
100 K	0.009	0.154	0.013	99.823

Table 3 Percentage of running time for each kernel of the Form A on double precision

Size	Kernel 1	Kernel 2	Kernel 3	Kernel 4
1 K	0.746	1.939	0.522	96.644
5 K	0.070	0.569	0.070	99.256
10 K	0.021	0.227	0.023	99.724
50 K	0.004	0.066	0.005	99.923
100 K	0.003	0.051	0.004	99.941

Table 4 Percentage of running time for each kernel of the Form B on double precision

Size	Kernel 1	Kernel 2	Kernel 3	Kernel 4
1 K	0.982	2.554	0.688	95.187
5 K	0.120	0.977	0.120	98.738
10 K	0.038	0.415	0.042	99.492
50 K	0.008	0.122	0.010	99.859
100 K	0.005	0.089	0.007	99.898

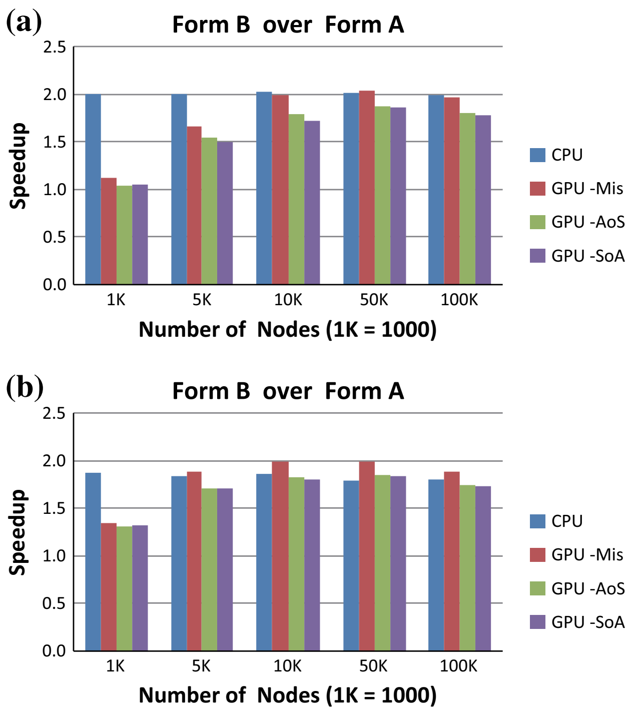


Fig. 9 Speedups of Form B over Form A on single and double precision. **a** Single precision, **b** double precision

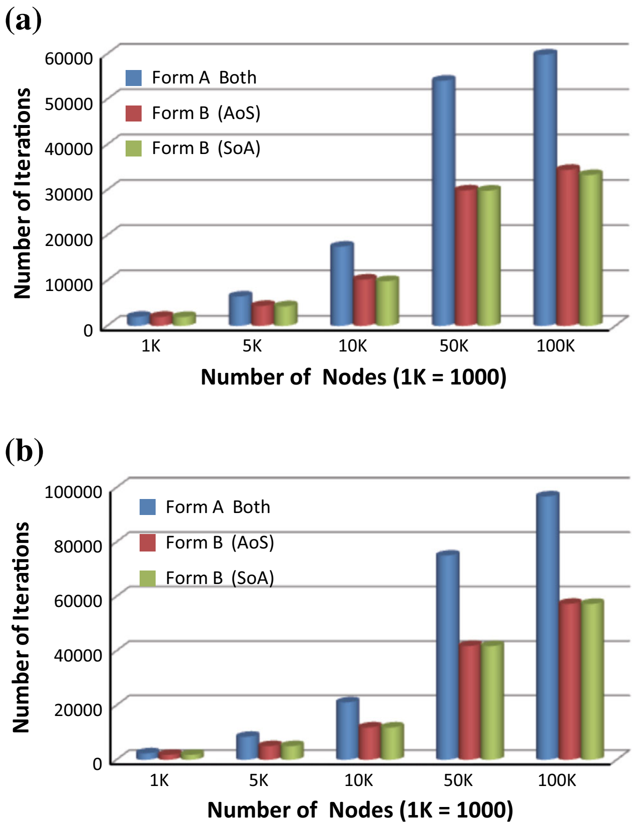


Fig. 10 Numbers of iterations until converging in Laplacian smoothing. **a** Single precision, **b** double precision

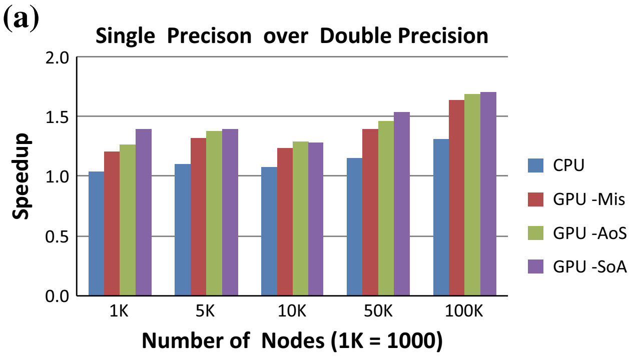
Form A needs much more iterations for converging. the numbers of iterations until converging are counted for the two forms on both single precision and double precision (see Fig. 10). It has been found that: for the Form A, the numbers of iterations in the two GPU versions implemented using the AoS and SoA data structures are exactly the same, while for the Form B the numbers of iteration are almost the same. However, the number of iterations in the Form A is always larger than that in the Form B. This is one of the reasons why the Form A is slower than the Form B. Therefore, the Form B is recommended in practical applications.

5.2.4 Performance on Single Precision and Double Precision

Considering the performances of GPU implementations on single and double precision, the GPU implementations on single precision are by nature faster than those performed on double precision. More specifically, both the Form A and Form B of the standard Laplacian smoothing implemented on single precision can achieve about 1.25×–1.5× speedups over their counterparts on double precision; see Fig. 11.

In practical applications, the GPU implementations developed on double precision are recommended to be adopted in the case when computational accuracy is the first key issue that needs to be considered. It has been demonstrated that the efficiency on double precision is worse than that on single precision, but this disadvantage is not obvious. When effi-

(a)



(b)

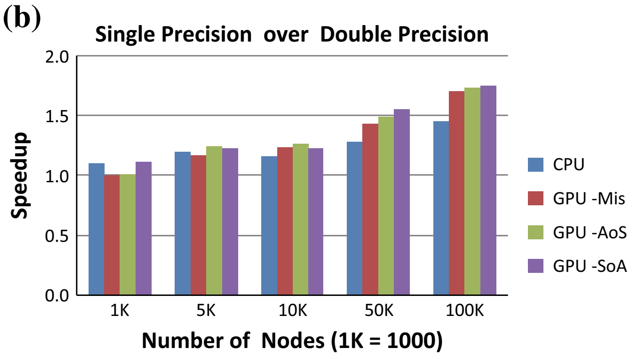


Fig. 11 Speedups of implementations on single precision over those on double precision. **a** Form A of LS, **b** form B of LS

```
#define REAL float/double
struct CVert {
  REAL x, y;          // Coordinates
  int nNeig;          // Number of neig.
  int * neig;         // Indices of neig.
  bool bBoundary;     // Indicator
};
struct CTrgl {
  int vIDs[3];        // Indices of vert.
};
struct CMesh {
  int nVert, nTrgl;
  CVert * verts;
  CTrgl * trgls;
};
```

Listing 1 Data structures for representing triangular meshes on the CPU

ciency is the most important issue that needs to be considered firstly, implementing the Laplacian smoothing on single precision is the better choice.

5.2.5 Outlook

In this paper, only the standard Laplacian smoothing has been implemented to illustrate the working process of the generic paradigm proposed for accelerating Laplacian-based mesh smoothing. In the future, more implementations of the variations such as weighted Laplacian smoothing and smart Laplacian smoothing are planned to be implemented. These variations are easy to implement according the process of the

```
#define REAL float/double
struct cuVert_MIS {
  REAL x, y;          // Coordinates
  int nNeig;          // Number of neig.
  int neig[64];       // Indices of neig.
  bool bBoundary;     // Indicator
};
struct cuTrgl_MIS {
  int vIDs[3];        // Indices of vert.
};
struct cuMesh_MIS {
  int nVert, nTrgl;
  cuVert * verts;
  cuTrgl * trgls;
};
```

Listing 2 Data structures represented by mis-aligned AoS

```
struct __align__(16) cuVert_AOS {
  float x, y;       // Coordinates
  int nNeig;        // Number of neig.
  bool bBoundary;   // Indicator
};
struct cuNeig_AOS {
  int neig[64];     // Indices of neig.
};
struct __align__(16) cuTrgl_AOS {
  int vIDs[3];      // Indices of vert.
};
struct cuMesh_AOS {
  int nVert, nTrgl;
  cuVert_AOS * verts;
  cuNeig_AOS * neigs;
  cuTrgl_AOS * trgls;
};
```

Listing 3 Data structures represented by aligned AoS on single precision

```
struct __align__(16) cuCoor_AOS {
  double x, y;      // Coordinates
};
struct __align__(8) cuInfo_AOS {
  int nNeig;        // Number of neig.
  bool bBoundary;   // Indicator
};
struct cuMesh_AOS_double {
  int nVert, nTrgl;
  cuCoor_AOS * coors;
  cuInfo_AOS * infos;
  cuNeig_AOS * neigs;
  cuTrgl_AOS * trgls;
};
```

Listing 4 Data structures represented by aligned AoS on double precision

generic paradigm; and only several minor modifications are needed on the basis of the implementations of the standard Laplacian smoothing.

In addition, much more work needs to be carried out for generalizing the proposed paradigm to smoothing surface meshes and volume meshes. The first obstacle is how to preserve the shape and volume features of mesh objects when

```
#define REAL float/double
struct cuVert_SOA {
  REAL *x, *y;       // Coordinates
  int *nNeig;        // Number of neig.
  int *neig;         // Indices of neig.
  bool *bBoundary;   // Indicator
};
struct cuTrgl_SOA {
  int *vIDs;         // Indices of vert.
};
struct cuMesh_SOA {
  int nVert, nTrgl;
  cuVert_SOA verts;
  cuTrgl_SOA trgls;
};
```

Listing 5 Data structures represented by SoA

smoothing. Another key issue is that it is needed to design efficient mesh data structures for specific type of meshes.

However, there are two rules that are possibly still useful when generalizing the proposed paradigm: (1) the Form B of Laplacian smoothing is faster than the Form A; (2) the mesh data structures represented by AoS layout achieve the best efficiency. When implementing Laplacian-based mesh smoothing for optimizing surface meshes or volume meshes, the above rules are needed to be demonstrated with real-world applications.

6 Conclusion

A generic paradigm has been developed for accelerating the Laplacian-based mesh smoothing on the GPU. Two forms of the standard Laplacian smoothing have been implemented using different mesh data structures on both single precision and double precision to demonstrate the working process of the proposed paradigm. Experimental results have indicated that: on both single precision and double precision, the GPU implementations developed using the data structures represented by aligned AoS layout achieve the best efficiency. It also has been observed that: for the two forms of standard Laplacian smoothing, the Form A that needs to swap intermediate nodal coordinates is always slower than the Form B that does not swap data. Thus, the first choice of data layouts for representing mesh data structures in Laplacian smoothing is the layout aligned AoS. And, the Form B of the standard Laplacian smoothing is suggested to be accepted in practical applications. The presented generic paradigm is effective and can help to guide the GPU implementations of Laplacian-based mesh smoothing in practice.

Acknowledgments The authors are grateful to the anonymous referee for helpful comments that improved this paper. This research was supported by the Natural Science Foundation of China (Grant Nos. 40602037 and 40872183).

References

1. Crespo, A.C.; Dominguez, J.M.; Barreiro, A.; Gomez-Gesteira, M.; Rogers, B.D.: GPUs, a new tool of acceleration in CFD: efficiency and reliability on smoothed particle hydrodynamics methods. PLoS One **6**(6), e20685 (2011)
2. Giles, M.B.; Mudalige, G.R.; Spencer, B.; Bertolli, C.; Reguly, I.: Designing OP2 for GPU architectures. J. Parallel Distrib. Comput. **73**(11), 1451–1460 (2013)
3. Qi, M.; Cao, T.T.; Tan, T.S.: Computing 2D constrained Delaunay triangulation using the GPU. IEEE T. Vis. Comput. Graph. **19**(5), 736–748 (2013)
4. Shuai, L.; Guo, X.H.; Jin, M.: GPU-based computation of discrete periodic centroidal Voronoi tessellation in hyperbolic space. Comput. Aided Des. **45**(2), 463–472 (2013)
5. D'Amato, J.P.; Venere, M.: A CPU–GPU framework for optimizing the quality of large meshes. J. Parallel Distrib. Comput. **73**(8), 1127–1134 (2013)
6. Zegard, T.; Paulino, G.H.: Toward GPU accelerated topology optimization on unstructured meshes. Struct. Multidiscip. Optim. **48**(3), 473–485 (2013)
7. NVIDIA: CUDA C Programming Guide v5.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (2013)
8. Munshi, A.: The OpenCL Specification, v2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf (2013)
9. Abal Abas, Z.; Salleh, S.; Manan, Z.: Extended advancing front technique for the initial triangular mesh construction on a single coil for radiative heat transfer. Arab. J. Sci. Eng. **38**(9), 2245–2262 (2013)
10. Freitag, L.; Ollivier-Gooch, C.: A cost/benefit analysis of simplicial mesh improvement techniques as measured by solution efficiency. Int. J. Comput. Geom. Appl. **10**(04), 361–382 (2000)
11. Shewchuk, J.R.: What is a good linear finite element? Interpolation, conditioning, anisotropy, and quality measures (preprint). University of California at Berkeley (2002)
12. Owen, S.J.: A survey of unstructured mesh generation technology. In: Proceedings of the 7th International Meshing Roundtable, vol. 3, pp. 239–267 (1998)
13. Freitag, L.; Jones, M.; Plassmann, P.: A parallel algorithm for mesh smoothing. SIAM J. Sci. Comput. **20**(6), 2023–2040 (1999)
14. Jiao, X.; Alexander, P.: Parallel Feature-Preserving Mesh Smoothing, Lecture Notes in Computer Science, vol. 3483, chap. 123, pp. 1180–1189. Springer, Berlin (2005)
15. Gorman, G.J.; Southern, J.; Farrell, P.E.; Piggott, M.D.; Rokos, G.; Kelly, P.H.J.: Hybrid openmp/mpi anisotropic mesh smoothing. Procedia Comput. Sci. **9**, 1513–1522 (2012)
16. Benłtez, D.; Rodrłguez, E.; Escobar, J.; Montenegro, R.: Performance Evaluation of a Parallel Algorithm for Simultaneous Untangling and Smoothing of Tetrahedral Meshes, chap. 32, pp. 579–598. Springer International Publishing (2014)
17. Sastry, S.; Shontz, S.: A parallel log-barrier method for mesh quality improvement and untangling. Eng. Comput. **30**(4), 503–515 (2014)
18. D'Amato, J.P.; Lotito, P.: Mesh optimization with volume preservation using GPU. Lat. Am. Appl. Res. **41**(3), 291–297 (2011)
19. Heuveline, V.; Lukarski, D.; Trost, N.; Weiss, J.P.: Parallel Smoothers for Matrix-Based Geometric Multigrid Methods on Locally Refined Meshes Using Multicore CPUs and GPUs, Lecture Notes in Computer Science, vol. 7174, chap. 14, pp. 158–171. Springer, Berlin (2012)
20. Monch, T.; Lawonn, K.; Kubisch, C.; Westermann, R.; Preim, B.: Interactive mesh smoothing for medical applications. Comput. Graph Forum **32**(8), 110–121 (2013)

21. Herrmann, L.R.: Laplacian-isoparametric grid generation scheme. J. Eng. Mech. Div. ASCE **102**(5), 749–907 (1976)
22. Blacker, T.D.; Stephenson, M.B.: Paving: a new approach to automated quadrilateral mesh generation. Int. J. Numer. Methods Eng. **32**(4), 811–847 (1991)
23. Vollmer, J.; Mencl, R.; Mller, H.: Improved Laplacian smoothing of noisy surface meshes. Comput. Graph Forum **18**(3), 131–138 (1999)
24. Freitag, L.A.: On combining Laplacian and optimization-based mesh smoothing techniques. In: Trends in Unstructured Mesh Generation, pp. 37–43 (1997)
25. Canann, S.A.; Tristano, J.R.; Staten, M.L.: An approach to combined Laplacian and optimization-based smoothing for triangular, quadrilateral, and quad-dominant meshes. In: Proceedings of 7th International Meshing Roundtable, pp. 479–494 (1998)
26. Farber, R.: CUDA Application Design and Development. Morgan Kaufmann (2011)
27. Bell, N.; Hoberock, J.: Thrust: Productivity-Oriented Library for CUDA, chap. 26, pp. 359–371. Morgan Kaufmann (2011)
28. Govender, N.; Wilke, D.N.; Kok, S.; Els, R.: Development of a convex polyhedral discrete element simulation framework for NVIDIA Kepler based GPUs. J. Comput. Appl. Math. **270**, 386–400 (2013)
29. Mudalige, G.R.; Giles, M.B.; Thiyagalingam, J.; Reguly, I.Z.; Bertolli, C.; Kelly, P.H.J.; Trefethen, A.E.: Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. Parallel Comput. **39**(11), 669–692 (2013)
30. Mistry, P.; Schaa, D.; Jang, B.; Kaeli, D.; Dvornik, A.; Meglan, D.: Data Structures and Transformations for Physically Based Simulation on a GPU, Lecture Notes in Computer Science, vol. 6449, chap. 17, pp. 162–171. Springer, Berlin (2011)
31. Strzodka, R.: Abstraction for AoS and SoA layout in C++, pp. 429–441. Morgan Kaufmann (2011)
32. Strzodka, R.: Data layout optimization for multi-valued containers in OpenCL. J. Parallel Distrib. Comput. **72**(9), 1073–1082 (2012)
33. Sung, I.J.; Liu, G.D.; Hwu, W.M.W.: DL: A data layout transformation system for heterogeneous computing. In: Innovative Parallel Computing (InPar), pp. 1–11. IEEE (2012)
34. Shewchuk, J.R.: Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In: Selected Papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering, FCRC ’96/WACG ’96, pp. 203–222. Springer, London (1996)