Int J Parallel Prog (2018) 46:963–991
https://doi.org/10.1007/s10766-017-0538-6

Performance Evaluation of GPU-Accelerated Spatial Interpolation Using Radial Basis Functions for Building Explicit Surfaces

Zengyu Ding[1] · **Gang Mei**[1] · **Salvatore Cuomo**[2] · **Nengxiong Xu**[1] · **Hong Tian**[3]

Received: 9 August 2017 / Accepted: 13 November 2017 / Published online: 20 November 2017

Abstract This paper focuses on evaluating the computational performance of parallel spatial interpolation with Radial Basis Functions (RBFs) that is developed by utilizing modern GPUs. The RBFs can be used in spatial interpolation to build explicit surfaces such as Discrete Elevation Models. When interpolating with large-size of data points and interpolated points for building explicit surfaces, the computational cost would be quite expensive. To improve the computational efficiency, we specifically develop a parallel RBF spatial interpolation algorithm on many-core GPUs, and compare it with the parallel version implemented on multi-core CPUs. Five groups of experimental tests are conducted on two machines to evaluate the computational efficiency of the presented GPU-accelerated RBF spatial interpolation algorithm. Experimental results indicate that: in most cases, the parallel RBF interpolation algorithm on many-core GPUs does not have any significant advantages over the parallel version on multi-core CPUs in terms of computational efficiency. This unsatisfied performance of the GPU-accelerated RBF interpolation algorithm is due to: (1) the limited size of global memory residing on the GPU, and (2) the need to solve a system of linear equations in each GPU thread to calculate the weights and prediction value of each interpolated point.

Keywords Spatial interpolation · Radial basis function (RBF) · Explicit surfaces · Parallel programming · Graphics processing unit (GPU)

✉ Gang Mei
gang.mei@cugb.edu.cn

[1] School of Engineering and Technology, China University of Geosciences, Beijing, China

[2] Department of Mathematics and Applications "R. Caccioppoli", University of Naples Federico II, Naples, Italy

[3] Faculty of Engineering, China University of Geosciences, Wuhan, China

1 Introduction

Spatial interpolation is the procedure for predicting the unknown value of a group of interpolated points according to the known value of a set of data points. The spatial interpolation is widely used in science and engineering applications, such as image processing [21], numerical analysis [32,35], geometrical computation [3], Geographic Information System (GIS) [16,17], Artificial Intelligence (AI) [22,37], and even Internet of Things (IoT) [4,7,19]. Several of the most frequently used spatial interpolation methods include: Inverse Distance Weighted Method (IDW) [33], Kriging method [26], Discrete Smoothing Interpolation method (DSI) [24,25], Moving Least Squares method (MLS) [30], and Radial Basis Functions (RBFs) Interpolation [8]. The performance of these interpolation methods has been excellently compared and analyzed by R. Franke [12].

The RBFs are commonly used to (1) approximate implicit surfaces in image processing and (2) build explicit surfaces such as Digital Elevation Model (DEM). The objective of approximating implicit surfaces with RBFs is different from that of building explicit surfaces with RBFs. In approximating implicit surfaces, the input data is a set of scattered points, and an implicit surface will be approximated, which attempts to fit the input scattered points. When building explicit surfaces using the RBF interpolation, a set of points with a specific type of known value is needed to calculate the unknown value of another set of interpolated points.

Much research work has been conducted to approximate implicit surfaces using the RBF approximation algorithms. For example, Cuomo et al. [8] analyzed theoretical and practical issues in using RBFs for reconstructing implicit curves and surfaces from point clouds. Hillier et al. [14] presented a generalized interpolation framework using RBFs to implicitly model three-dimensional continuous geological surfaces from scattered multivariate structural data. Macedo et al. [23] introduced the Hermite Radial Basis Function (HRBF) implicit method to calculate a global implicit function that can interpolate scattered multivariate Hermite data. Lin et al. [20] proposed a novel implicit surface reconstruction approach, named Dual-RBF.

Also, several efforts have been dedicated to building various types of explicit surfaces using the RBF interpolation algorithms. For example, Izquierdo et al. [18] proposed a new interpolation scheme for Compactly-Supported Radial Basis Functions (CS-RBFs) to address the problem of the interpolation of explicit surfaces with vertical faults from scattered data.

In many science or engineering applications such as those in the field of GIS and IoT, the involved data size could be quite large. When adopting the RBF interpolation method to deal with the large size of data sets, the computational cost would be quite expensive; especially the computational efficiency would be unsatisfied.

The techniques in HPC (High Performance Computing) are widely used to improve computational efficiency in various science and engineering applications, such as image processing [10,11,31], 3D data denoising [5], spatial interpolation [28,29], and numerical computation [6,27].

One of the effective strategies to solve the problem is to perform the RBF-based approximations or interpolations in parallel on various parallel computing platforms such as shared-memory computers, distributed-memory computers, or even clusters.

For example, Cuomo et al. [9] described a parallel implicit method based on RBFs for surface reconstruction by exploiting the Graphic Processor Units (GPUs) acceleration. Wang et al. [36] presented a parallel algorithm for RBF-based surface reconstruction from contours on multi-core CPUs. Yokota et al. [38] developed a parallel algorithm for RBF interpolation that exhibits $O(n)$ complexity, requires $O(n)$ storage, and scales excellently up to a thousand processes on powerful clusters.

To be best of the authors' knowledge, there is no previously reported work specifically focused on developing or evaluating GPU-accelerated spatial interpolation with RBFs for building explicit surfaces. The currently reported relevant work mainly aims at accelerating the approximation of implicit surfaces on the GPU [9].

In this paper, we specifically focus on evaluating the computational performance of GPU-accelerated spatial interpolation with RBFs. We first parallelize the RBF interpolation on many-core GPU and then compare it with the parallel version implemented on multi-core CPU. We also carry out five groups of experimental tests on two different machines to evaluate the computational efficiency.

The paper is organized as follows. Section 2 briefly introduces the spatial interpolation using RBFs. Section 3 concentrates mainly on our parallel implementations of the RBF interpolation on multi-core CPU and many-core GPU. Section 4 presents several experimental tests, and Section 5 discusses the results. Finally, Sect. 6 draws some conclusions.

2 Background: Spatial Interpolation Using RBFs

Given a set of N distinct points $\left(x_j, y_j\right),\ j = 1, \ldots, N$, where $x_j \in R^s$ and $y_j \in R$, the scattered data interpolation problem consists in finding an interpolant function on F such that:

$$F\left(x_j\right) = y_j,\ \ j = 1, \ldots, N. \tag{1}$$

In the univariate setting ($s = 1$), the interpolant F is usually chosen in a suitable function space. A common approach assumes the function F as a linear combination of certain basis functions Φ_j.

$$F(x) = \sum_{j=1}^{N} w_j \Phi_j(x) \tag{2}$$

In a multivariate setting $x_j \in R^s, s > 1$, the problem is more complex. In order to have a well-posed multivariate scattered data interpolation problem, it is not possible to fix in advance the basis $\{\Phi_1, \Phi_2, \ldots, \Phi_N\}$, since the basis functions must depend on the data sites x_j.

The data dependent space for RBF interpolation can be easily generated by means of the radial functions:

$$\Phi_j = \phi\left(\|x - x_j\|\right) \tag{3}$$

The points x_j to which the basic function ϕ is shifted are usually referred to as centers. While there may be circumstances that suggest choosing these centers different from the data sites one generally picks the centers to coincide with the data sites.

In fact, a practical interpolation problem consists of two sub problems: finding the interpolant F and evaluating it on an assigned set of points. The coefficients w_j in Eq. (2) are obtained by imposing the interpolation conditions (Eq. 1).

$$F\left(x_i\right) = \sum_{j=1}^{N} w_j \phi\left(\|x_i - x_j\|\right) = y_i, \quad i = 1, \ldots, N \tag{4}$$

This leads to solving the linear system of equations $Ax = b$ in Eq. (5).

$$\begin{aligned} Aw &= B \\ A &= \begin{bmatrix} \phi\left(\|x_1 - x_1\|\right) & \phi\left(\|x_1 - x_2\|\right) & \cdots & \phi\left(\|x_1 - x_N\|\right) \\ \phi\left(\|x_2 - x_1\|\right) & \phi\left(\|x_2 - x_2\|\right) & \cdots & \phi\left(\|x_2 - x_N\|\right) \\ \vdots & \vdots & \ddots & \vdots \\ \phi\left(\|x_N - x_1\|\right) & \phi\left(\|x_N - x_1\|\right) & \cdots & \phi\left(\|x_N - x_N\|\right) \end{bmatrix} \\ w &= \left[w_1, w_2, \ldots, w_N\right]^T, \quad B = \left[y_1, y_2, \ldots, y_N\right]^T \end{aligned} \tag{5}$$

Given a set of M points $\xi = \{\xi_1, \xi_2, \ldots, \xi_M\}$, the evaluation of the interpolant F on ξ can be computed as a matrix-vector product (Eq. 6).

$$F\left(\xi_i\right) = \sum_{j=1}^{N} w_j \phi\left(\xi_i - x_j\right), \quad i = 1, 2, \ldots, M \tag{6}$$

It is well known that in order to have a well-posed problem (Eq. 5), the matrix A must be nonsingular. Unfortunately, a complete characterization of the class of all basic functions ϕ that generate a nonsingular matrix for an arbitrary set $\chi = \{x_1, x_2, \ldots, x_N\}$ of distinct data points is still lacking. The situation gets better in the case of positive definite matrices, which are always non-singular. Popular RBFs ϕ_i, that give rise to positive definite interpolation matrices, are summarized as follows.

Multi-quadrics (MQ):

$$\phi_j\left(x\right) = \sqrt{\left(c^2 + \|x - x_j\|^2\right)}, \tag{7}$$

Inverse MQ (IMQ):

$$\phi_j\left(x\right) = 1/\sqrt{\left(c^2 + \|x - x_j\|^2\right)}, \tag{8}$$

Thin-plate splines (TPS):

$$\phi_j\left(x\right) = \|x - x_j\|^2 \ln\left(\|x - x_j\|/c\right), \tag{9}$$

Gaussians:

$$\phi_j\left(x\right) = \exp\left(-c\|x - x_j\|^2\right), \tag{10}$$

where c is the shape parameter, which can be selected according to the suggestions given in the literature [13,34,35].

3 Methods: GPU-Accelerated Spatial Interpolation Using RBFs for Building Explicit Surfaces

In this section, we will first introduce our basic ideas behind the presented GPU-accelerated spatial interpolation using RBFs for building explicit surfaces, and then describe the details of three implementations, i.e., (1) the serial implementation of the spatial interpolation using RBFs, (2) the parallel implementation developed on multi-core CPU, and (3) the parallel implementation by utilizing a single many-core GPU.

3.1 Basic Ideas Behind the GPU-Accelerated Spatial Interpolation Using RBFs

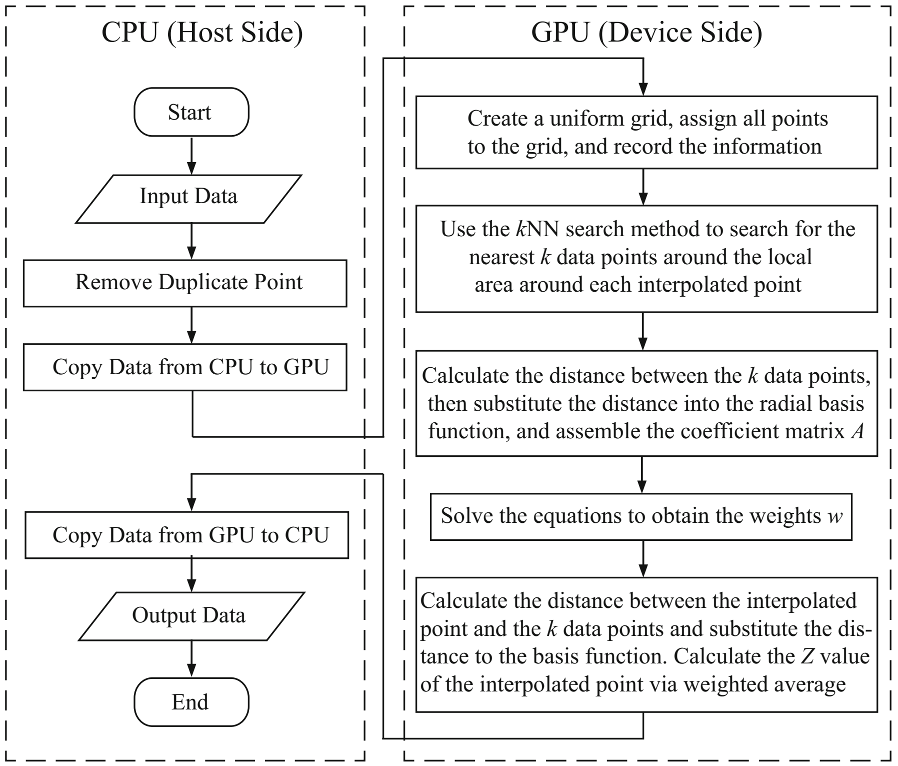
3.1.1 Overview

The spatial interpolation algorithm using RBFs for building explicit surfaces is inherently suitable to be parallelized on GPU architecture. This is because that: in the RBF-based interpolation algorithm, the desired prediction value for each interpolated point can be calculated independently, which means that it is naturally to calculate the prediction values for many interpolated points concurrently without any data dependencies between the interpolating procedures for any pair of the interpolated points.

Due to the inherent feature of the RBF-based spatial interpolation algorithm, it is allowed a single thread to calculate the interpolation value for an interpolated point. For example, assuming there are n interpolation points that are needed to be predicted their values such as elevations, and then it is needed to allocated n threads to concurrently calculate the desired prediction values for all n interpolated points. Therefore, the RBF-based spatial interpolation method is quite suitable to be parallelized on GPU architecture.

In the RBF-based spatial interpolation, there are two choices for determining the region of data points for each interpolated point. The first is to use all the data points to calculate the prediction value of each interpolated points, while the second is to employ a local set of data points to evaluate the prediction value of an interpolated point. The interpolation methods adopting the first choice of selecting a global set of data points are referred to as *Global* interpolation, while those interpolation methods employing the second opinion are called *Local* interpolation.

In the presented RBF-based spatial interpolation for building explicit surfaces, we adopt the second choice to determine the region of data points for each interpolated point. That is, we only use the local set of data points around an interpolated point to calculate the prediction value. The local set of data points for each interpolated is found using the k Nearest Neighbors algorithm (kNN) [29]. Moreover, we adopt the Globally-Supported RBFs (GS-RBFs) rather than the Compactly-Supported RBFs (CS-RBFs) for the local set of data points to compute the prediction value of each interpolated point.



CPU (Host Side)
GPU (Device Side)
Start
Input Data
Remove Duplicate Point
Copy Data from CPU to GPU
Copy Data from GPU to CPU
Output Data
End
Create a uniform grid, assign all points to the grid, and record the information
Use the *k*NN search method to search for the nearest *k* data points around the local area around each interpolated point
Calculate the distance between the *k* data points, then substitute the distance into the radial basis function, and assemble the coefficient matrix *A*
Solve the equations to obtain the weights *w*
Calculate the distance between the interpolated point and the *k* data points and substitute the distance to the basis function. Calculate the *Z* value of the interpolated point via weighted average

Fig. 1 The procedure of the presented GPU-accelerated spatial interpolation algorithm using RBFs

In summary, there are two key ideas behind the presented RBF-based spatial interpolation algorithm for constructing explicit surfaces:

(1) We use a local set of data points around each interpolated point to calculate the prediction value of the interpolated point. The local set of data points is found using a *k*NN algorithm.
(2) We employ GS-RBFs for the local set of data points to compute the prediction value of the interpolated point.

The process of the presented GPU-accelerated spatial interpolation algorithm using RBFs is illustrated in Fig. 1. First, the input data is stored on the host side and then transferred to the device side. Second, on the device side, an even grid is created to help conduct the *k*NN search procedure. Third, the local set of data points for each interpolated point is found using the *k*NN algorithm; and then the distances between the data points are calculated, after that the coefficient matrix can be formed according to the selected GS-RBFs. Finally, weights can be obtained by solving the linear equations, and the desired prediction value for each interpolated points can be achieved by weighting average using those weights.

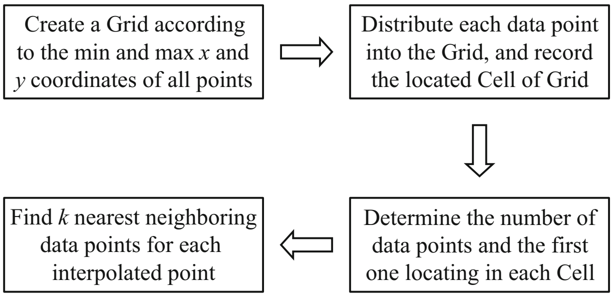

Create a Grid according to the min and max x and y coordinates of all points
Distribute each data point into the Grid, and record the located Cell of Grid
Determine the number of data points and the first one locating in each Cell
Find k nearest neighboring data points for each interpolated point

Fig. 2 The procedure of the *k*NN search for finding a local set of data points for each interpolated point

3.1.2 Stage 1: The kNN Search

In the RBF-based spatial interpolation algorithm, it is needed to find the *k* nearest neighboring data points for each interpolated point. The *k*NN search algorithm is directly derived from our previous work [29]. The procedure of the *k*NN search is illustrated in Fig. 2, and more details are described as follows.

Step 1: Creating an even grid

The creating of an even planar grid is straightforward. We first determine the planar rectangular region for partitioning by finding the minimum and maximum *x* and *y* coordinates of all points. Then, the numbers of rows and columns of the grid can be easily determined by dividing the rectangle with the width of the square cell; see a simple illustration in Fig. 3.

Step 2: Distributing data points into cells

The objective of distributing all data points into the grid cells is to find out in which grid cell each data point is located. The distributing of each data point is in fact to determine the row and column indices of the cell in which it locates. Since that the grid cells are indexed sequentially first by rows and then by columns, the procedure of distributing can be easily carried out. First, the differences between the coordinates of a data point and the minimum coordinates of all cells are calculated; then the indices of column and row can be determined by dividing the above differences with the cell width.

Step 3: Determining data points in each cell

The objective of this step is to determine the number and the indices of those data points located in the same cell. The number of data points located in the same cell can be determined with the use of a *segmented* parallel reduction. After sorting all data points according to cell indices, the data points are sequentially stored in a group of segments; each segment is flagged with the cell index, and contains the indices of data points locating in the same cell. The number of those data points located in the same cell can be obtained by performing a reduction for each segment; see Fig. 4a. Moreover, the head index of the first point of each segment can be determined using segmented parallel scan; see Fig. 4b.

Step 4: Searching nearest neighbors

The process of *k*NN search for each interpolated point can be summarized as the following substeps: (1) locating the interpolate point into the even grid, (2) determining

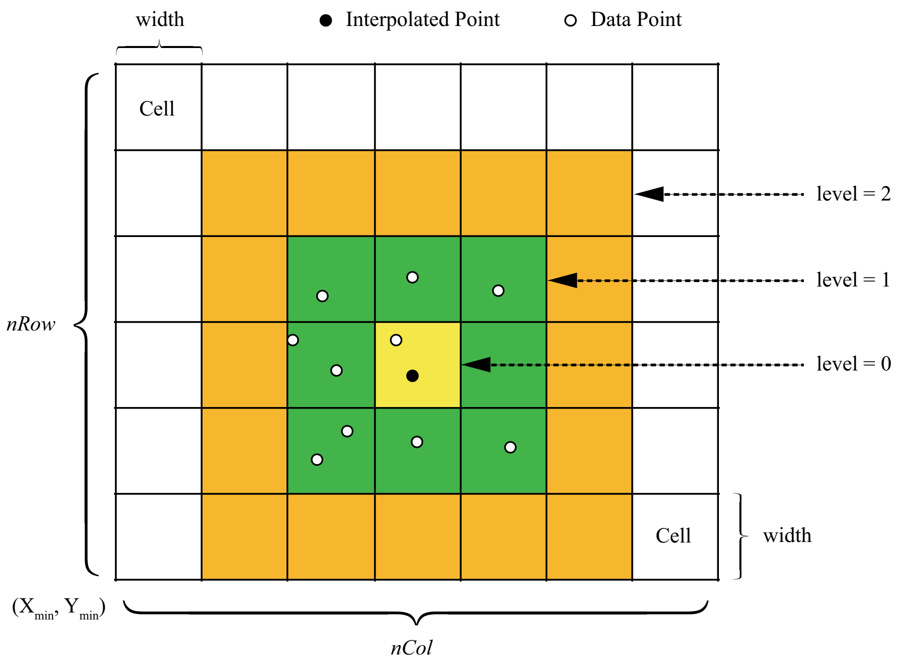


Fig. 3 The creation of an even grid according to the minimum and maximum coordinates of all the data points and interpolated points (This figure is directly derived from our previous work [29])

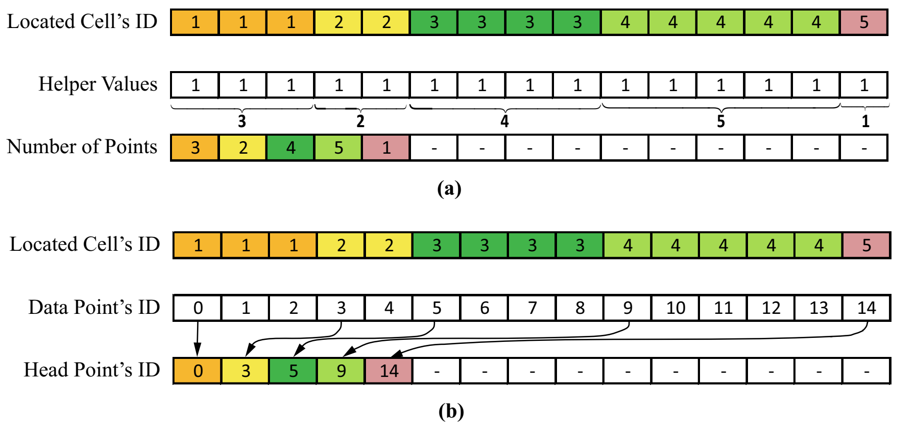


Fig. 4 Demonstration of determining the number of data points distributed in each cell and the index of the head point. **a** The number of points; **b** the index of the head point (This figure is directly derived from our previous work [29])

the level of cell expanding (see Fig. 3), and (3) finding the *k* nearest neighbors within the local region. More details on searching the nearest neighboring data points for each interpolated points are presented in our previous work [29].

3.1.3 Stage 2: The Interpolating Using RBFs

After finding the nearest neighbors of each interpolated point using the above-described *k*NN search algorithm, the distances (1) between any pair of the nearest neighboring data points and (2) between each nearest neighbor and the interpolated point can be calculated; after that, the coefficient matrix can be formed according to the selected GS-RBFs, i.e., the Multi-quadrics RBF (see Eq. 7). Finally, the weights can be obtained by solving the equations (see Eq. 5), and the desired prediction value for each interpolated points can be achieved by weighting average using those weights (see Eq. 6).

3.2 Sequential and Parallel Implementations

In this subsection, we will introduce details of the following three implementations, i.e., (1) the sequential implementation of the spatial interpolation using RBFs, (2) the parallel implementation developed on multi-core CPU, and (3) the parallel implementation by utilizing a many-core GPU. Our focus in this work is to evaluate the computational performance of the GPU-accelerated spatial interpolation algorithm using RBFs. Thus, we also implement the sequential version and the parallel version on multi-core CPU, and then compare the GPU version with that of the sequential version and the parallel version on multi-core CPU.

It should be also noted that the source code of the above three implementations is publicly available at: https://figshare.com/s/e9a2fc20daa963097d1d.

3.2.1 Sequential Implementation

There are two major sub-procedures in the sequential implementation. The first is to create the even grid and then record the number and indices of those data points located in each grid cell. The second is to loop over all interpolated points to first find the *k*NN of each interpolated point and then calculate the prediction value based on RBFs.

In the sequential implementation, we first sequentially loop over all points to determine the planar rectangular region for partitioning by finding the minimum and maximum x and y coordinates of all points. Then, the numbers of rows and columns of the grid can be quite easily determined by dividing the rectangle with the width of the square cell. After that, we sequentially distribute all the data points into the grid cells and record the index of the located cell for each data point. Then, we sort all the data points according to the index of the cell it locates ascendingly using the `std::sort()` function, and sequentially loop over all the sorted data points again to directly record the number and indices of data point located in the same grid cell. Finally, we loop over all interpolated points to first find the *k*NN of each interpolated point and then calculate the prediction value based on RBFs.

3.2.2 Parallel Implementation on Multi-core CPU

There are two major sub-procedures in the presented parallel spatial interpolation algorithm using RBFs. The first is to create the even grid and then record the number and indices of those data points located in each grid cell. The second is to loop over all interpolated points to first find the kNN of each interpolated point and then calculate the prediction value based on RBFs.

The first sub-procedure is implemented by strongly utilizing the *Thrust* library [1,2]. Thrust is a C++ template library for parallel platforms based on the Standard Template Library (STL). Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce. Thrust allows users to implement high-performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with technologies such as C++, CUDA, OpenMP, and TBB [15].

In the first sub-procedure, we first employ the function `thrust::minmax_element()` to find the boundary range of all input data points and interpolated points, i.e., to determine the minimum and maximum x and y coordinates of all data points and interpolated points. Then, we use the function `thrust::sort_by_key` to sort all data points ascendingly according to their coordinates. Finally, we employ the function `thrust::unique_by_key` to find the head indices of the data points located in each grid cell, and adopt the function `thrust::reduce_by_key` to determine the number of data points residing in each grid cell.

The second sub-procedure is parallelized by exploiting the interface OpenMP with the simple use of the OpenMP directive “`#pragma omp parallel for`”. In the second sub-procedure, a loop needs to be performed over all the interpolated points to first find the k nearest neighboring data points for each interpolated point and then calculate the nodal distances, coefficients, and weights. Due to the fact that there are no data dependencies between the calculating of prediction values of any pair of interpolated points, the spatial interpolating for all interpolated points can be parallelized by simply adding the OpenMP directive “`#pragma omp parallel for`” before the `for_loop`.

3.2.3 Parallel Implementation on Many-Core GPU

As introduced in the above subsection, there are two major sub-procedures in the presented parallel spatial interpolation algorithm using RBFs. The first is to create the even grid and then record the number and indices of those data points located in each grid cell. The second is to loop over all interpolated points to first find the kNN of each interpolated point and then calculate the prediction value based on RBFs.

The first sub-procedure is implemented by strongly utilizing the *Thrust* library [1,2]. For that the Thrust library is fully interoperable with technologies such as C++, CUDA, OpenMP, and TBB [15], these employed functions provided by Thrust library, such as `thrust::minmax_element()`, `thrust::sort_by_key`, `thrust::unique_by_key`, and `thrust::reduce_by_key`, can be executed in parallel on both the multi-core CPU and many-core GPU. Therefore, there is no need to modify those functions to port them from the multi-core CPU to the many-core GPU. It only needs to replace the container `thrust::host_vector` with the

Table 1 Specifications of the employed two workstations for performing the experimental tests

Specifications	PC NO.1	PC NO.2
CPU	Intel Xeon E5-2650 v3	Intel Xeon E5-2680 v2
CPU frequency	2.30 GHz	2.80 GHz
CPU RAM	144 GB	96 GB
CPU core	40	40
GPU	Quadro M5000	Tesla k40c
GPU memory	8 GB	12 GB
GPU core	2048	2880
OS	Windows 7 Professional	Windows 7 Professional
Compiler	Visual Studio 2010	Visual Studio 2010
CUDA version	v8.0	v8.0

container `thrust::device_vector`. After the modifying, the above-mentioned functions can be automatically executed in parallel on the GPU.

A specific CUDA kernel is designed for the second sub-procedure. In the kernel, each thread is invoked to calculate the prediction value for an interpolated point. More specifically, each thread is responsible to (1) find the k nearest neighboring data points for an interpolated point and (2) calculate the distances between the found k nearest data points, the corresponding coefficient matrix, the weights, and finally the desired prediction value for an interpolated point.

4 Experimental Results

4.1 Experimental Environment and Testing Data

4.1.1 Experimental Environment

To evaluate the computational performance of the presented RBF-based spatial interpolation algorithm, we carry out five groups of experimental tests on two different workstations. The specifications of the employed workstations are listed in Table 1.

4.1.2 Testing Data

We have created five groups of testing data (see Table 2). In each group of testing data, a set of input data points that the x, y, and z coordinates are known before interpolating and a set of interpolated points that only the x and y coordinates are known while the z coordinate is intended to be predicted using the presented RBF-based interpolation algorithm.

Each set of data points are created by randomly distributing on a parametric surface; see Fig. 5. The equation of the parametric surface is demonstrated in Eq. (11). More specifically, both of the x and y coordinates are randomly generated in the range of

Table 2 The used five groups of experimental testing data

Data set	Num. of data points	Num. of interpolated points
Size 1	69,198	72,301
Size 2	136,960	144,601
Size 3	276,991	287,977
Size 4	530,870	580,194
Size 5	963,895	1,149,231

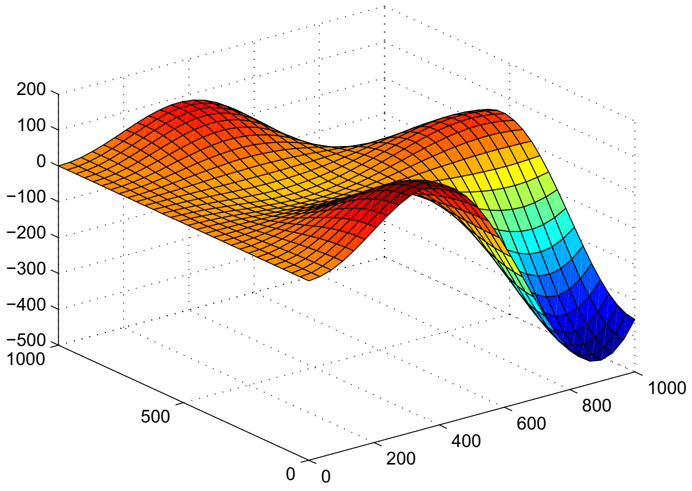


Fig. 5 The parametric surface for generating input experimental testing data

0—1000, while the z coordinate is simply calculated according to Eq. (11) after the x and y coordinates have been determined.

$$z = 1000 * \sin\left(\frac{\pi x}{6 * 1000}\right) \sin\left(\frac{7 * \pi x}{4 * 1000}\right) \cos\left(\frac{3 * \pi y}{4 * 1000}\right) \cos\left(\frac{5 * \pi y}{4 * 1000}\right), \quad 0 \leq x, y \leq 1000 \tag{11}$$

The generation of five sets of interpolated points is the same as that of the data points. Both of the x and y coordinates of each interpolated points are randomly generated in the range of 0–1000. However, the z coordinate is not needed to be calculated according to Eq. (11).

4.2 Running Time and Speedup

In the presented RBF-based spatial interpolation algorithm, we use a local set of data points around an interpolated point to calculate the prediction value of the interpo-

Table 3 Experimental results of five groups of testing data when $k = 10$ on the PC NO.1

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	375	312	75	1.20	5.00
Size 2	764	601	129	1.27	5.92
Size 3	1529	1194	223	1.28	6.86
Size 4	3112	2377	386	1.31	8.06
Size 5	5959	4789	686	1.24	8.69

Table 4 Experimental results of five groups of testing data when $k = 20$ on the PC NO.1

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	1287	328	142	3.92	9.06
Size 2	2574	624	216	4.13	11.92
Size 3	5140	1233	397	4.17	12.95
Size 4	10,327	2403	782	4.30	13.21
Size 5	20,241	4813	1525	4.21	13.27

lated point. The local set of data points is determined by employing the kNN search procedure. The size of the local set of data points can be user-specified by fixing the value of k.

In our experimental tests, five values of k are specified, i.e., 10, 20, 40, 60, and 80. For each of the five different values of k, the five groups of experimental tests are performed on both of two workstations. The running time and corresponding speedup of each group of experimental tests are presented in the following section.

4.2.1 Experimental Tests on the PC NO.1

On the workstation featured with the GPU M5000, five groups of experimental tests are conducted with five different values of k. The running time and corresponding speedups of each group of experimental tests are listed in Tables 3, 4, 5, 6 and 7.

The experimental results indicate that: (1) for the same group of testing data, with the increase of the value of k, the speedups achieved by the two parallel versions of the RBF-based spatial interpolation algorithm become large. (2) for the same value of k, with the increase of the size of testing data, the speedups obtained by the parallel implementations on multi-core CPU and many-core GPU also become large.

4.2.2 Experimental Tests on the PC NO.2

On the workstation featured with the GPU K40c, five groups of experimental tests are conducted with five different values of k. The running time and corresponding speedups of each group of experimental tests are listed in Tables 8, 9, 10, 11 and 12.

Table 5 Experimental results of five groups of testing data when $k = 40$ on the PC NO.1

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	6108	655	593	9.33	10.30
Size 2	12,230	1123	1071	10.89	11.42
Size 3	24,336	2036	1978	11.95	12.30
Size 4	49,015	3619	3932	13.54	12.47
Size 5	97,235	6965	7935	13.96	12.25

Table 6 Experimental results of five groups of testing data when $k = 60$ on the PC NO.1

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	16,325	1458	1299	11.20	12.57
Size 2	32,698	2636	2467	12.40	13.25
Size 3	65,099	5125	4663	12.70	13.96
Size 4	130,790	9118	9451	14.34	13.84
Size 5	258,696	17,285	18,850	14.97	13.72

Table 7 Experimental results of five groups of testing data when $k = 80$ on the PC NO.1

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	35,022	2855	3526	12.27	9.93
Size 2	69,919	5312	6846	13.16	10.21
Size 3	139,371	9329	13,378	14.94	10.42
Size 4	280,457	18,510	26,905	15.15	10.42
Size 5	555,049	35,924	53,768	15.45	10.32

Table 8 Experimental results of five groups of testing data when $k = 10$ on the PC NO.2

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	375	258	83	1.45	4.52
Size 2	671	523	145	1.28	4.63
Size 3	1365	1030	260	1.33	5.25
Size 4	2769	2004	515	1.38	5.38
Size 5	5359	4048	988	1.32	5.42

Table 9 Experimental results of five groups of testing data when $k = 20$ on the PC NO.2

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	1116	305	144	3.66	7.75
Size 2	2262	546	265	4.14	8.54
Size 3	4516	1100	500	4.11	9.03
Size 4	9072	2098	1003	4.32	9.04
Size 5	17,932	4228	1999	4.24	8.97

Table 10 Experimental results of five groups of testing data when $k = 40$ on the PC NO.2

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	5297	616	715	8.60	7.41
Size 2	10,639	913	1400	11.65	7.60
Size 3	21,153	1630	2728	12.98	7.75
Size 4	42,510	3042	5558	13.97	7.65
Size 5	84,115	5843	11,083	14.40	7.59

Table 11 Experimental results of five groups of testing data when $k = 60$ on the PC NO.2

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	14,844	1202	1690	12.35	8.78
Size 2	29,640	2106	3329	14.07	8.90
Size 3	59,248	3955	6494	14.98	9.12
Size 4	119,387	7792	13,189	15.32	9.05
Size 5	234,827	15,198	26,359	15.45	8.91

Table 12 Experimental results of five groups of testing data when $k = 80$ on the PC NO.2

Data set	Running time (/ms)			Speedup	
	Sequential	Parallel on CPU	Parallel on GPU	Parallel on CPU	Parallel on GPU
Size 1	31,574	2208	3717	14.30	8.49
Size 2	62,946	4243	7339	14.84	8.58
Size 3	125,564	8268	14,317	15.19	8.77
Size 4	253,407	16,435	28,954	15.42	8.75
Size 5	499,622	32,331	57,933	15.45	8.62

As the same as the two behaviors concluded from the experimental results performed on the workstation featured with the GPU M5000, the experimental results conducted on the workstation featured the GPU K40c also indicate that: (1) for the same group of testing data, with the increase of the value of k, the speedups achieved by the two parallel versions of the RBF-based spatial interpolation algorithm become large. (2) for the same value of k, with the increase of the size of testing data, the speedups obtained by the parallel implementations on multi-core CPU and many-core GPU also become large.

4.3 Interpolation Accuracy

One of the key issues in spatial interpolation is to evaluate the accuracy by comparing the interpolated results with the theoretically exact results or really observed results. However, it also should be noted that: in some cases, it is not able to evaluate the interpolation accuracy since the theoretically exact or the really observed results cannot be obtained.

In this work, we adopt the following two metrics for indicating computational errors to evaluate the interpolation accuracy.

Metric 1: Normalized Maximum Error (NME)

$$NME = \frac{1}{\max\limits_{1 \le i \le N_i} |z_a|} \max_{1 \le i \le N_i} |z_n - z_a| , \tag{12}$$

Metric 2: Normalized Root-Mean-Square Error (NRMSE)

$$NRMSE = \frac{1}{\max\limits_{1 \le i \le N_i} |z_a|} \sqrt{\frac{1}{N_i} \sum_{i=1}^{N_i} |z_n - z_a|^2}, \tag{13}$$

where N_i is the number of interpolated points, z_a is the theoretically exact solution of the ith interpolated point, which can be easily calculated according to Eq. (11); and z_n is the predicted value of the ith interpolated point.

The interpolation accuracy of the five groups of experimental tests is listed in Tables 13 and 14. Note that the data presented in Tables 13 and 14 is completely the same. However, to be easy to evaluate the impact of (1) the size of data set and (2) the value of k on the accuracy, we specifically organize the data in two different layouts of tables (Tables 13 and 14).

5 Discussion

In this section, we will analyze (1) the impact of the size of data set and the value of k on the computational efficiency of the presented RBF-based spatial interpolation algorithm, (2) the impact of the size of data set and the value of k on the interpolation

Table 13 Interpolation accuracy of the GPU-accelerated spatial interpolation algorithm when fixing the value of k

Data set	Numerical error	Value of k				
		10	20	40	60	80
Size 1	NME	1.23E−03	6.82E−04	1.01E−03	1.23E−03	1.65E−03
	NRMSE	3.75E−05	4.19E−05	5.80E−05	7.40E−05	9.14E−05
Size 2	NME	5.54E−04	3.03E−04	4.34E−04	5.39E−04	7.40E−04
	NRMSE	1.84E−05	2.07E−05	2.82E−05	3.58E−05	4.33E−05
Size 3	NME	2.51E−04	1.31E−04	2.23E−04	2.50E−04	3.23E−04
	NRMSE	8.89E−06	9.88E−06	1.34E−05	1.69E−05	2.07E−05
Size 4	NME	1.15E−04	7.70E−05	1.31E−04	1.60E−04	1.89E−04
	NRMSE	4.52E−06	5.06E−06	6.84E−06	8.71E−06	1.08E−05
Size 5	NME	8.46E−05	5.91E−05	8.04E−05	8.43E−05	1.26E−04
	NRMSE	2.44E−06	2.72E−06	3.72E−06	4.91E−06	6.31E−06

Table 14 Interpolation accuracy of the GPU-accelerated spatial interpolation algorithm when fixing the size of data sets

Value of k	Numerical error	Data set				
		Size 1	Size 2	Size 3	Size 4	Size 5
10	NME	1.23E−03	5.54E−04	2.51E−04	1.15E−04	8.46E−05
	NRMSE	3.75E−05	1.84E−05	8.89E−06	4.52E−06	2.44E−06
20	NME	6.82E−04	3.03E−04	1.31E−04	7.70E−05	5.91E−05
	NRMSE	4.19E−05	2.07E−05	9.88E−06	5.06E−06	2.72E−06
40	NME	1.01E−03	4.34E−04	2.23E−04	1.31E−04	8.04E−05
	NRMSE	5.80E−05	2.82E−05	1.34E−05	6.84E−06	3.72E−06
60	NME	1.23E−03	5.39E−04	2.50E−04	1.60E−04	8.43E−05
	NRMSE	7.40E−05	3.58E−05	1.69E−05	8.71E−06	4.91E−06
80	NME	1.65E−03	7.40E−04	3.23E−04	1.89E−04	1.26E−04
	NRMSE	9.14E−05	4.33E−05	2.07E−05	1.08E−05	6.31E−06

accuracy, and (3) the key factors that limit the efficiency of the GPU-accelerated spatial algorithm using RBFs. Furthermore, we point out our potential future work that would improve the proposed algorithm in this work.

5.1 Impact of the Size of Data Set and the Value of *k* on the Computational Efficiency

In this subsection, we will discuss the impact of the size of data sets (i.e., the input data points and the interpolated points) and the different values of k on the computational efficiency of the proposed RBF-bases spatial interpolation algorithm for building explicit surfaces.

5.1.1 Impact of the Size of Data Set on the Computational Efficiency

We have conducted five groups of experimental tests on two machines. To evaluate the impact the size of data set on the computational efficiency, we specifically re-organize the experimental results by fixing the value of k for five different sizes of data set; see Fig. 6.

The experimental results illustrated in Fig. 6 clearly indicate that: with the increase of the size of data sets, the speedups achieved by the parallel RBF-based spatial interpolation algorithm on the GPU stably become large. This behavior is quite reasonable since with the increase in the data size, the computational cost needed to perform the spatial interpolation is thus correspondingly increased. Moreover, in this case, the power of the GPU parallelization is highly exploited.

In short, when accelerating a relatively small size of data sets on the GPU, the power of the parallelism on the GPU cannot be fully utilized. For but the large size of data sets, the advantage in parallelizing on the GPU becomes more obvious and thus the achieved speedups become larger.

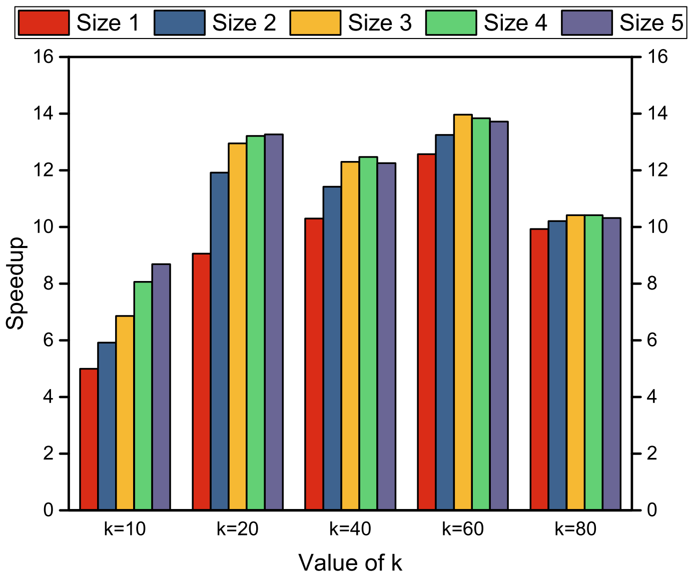
5.1.2 Impact of the Value of k on the Computational Efficiency

Similar to evaluating the impact of data size on the computational efficiency, we also specifically re-organize the experimental results by fixing the sizes of data sizes for five different values of k; see Fig. 7.

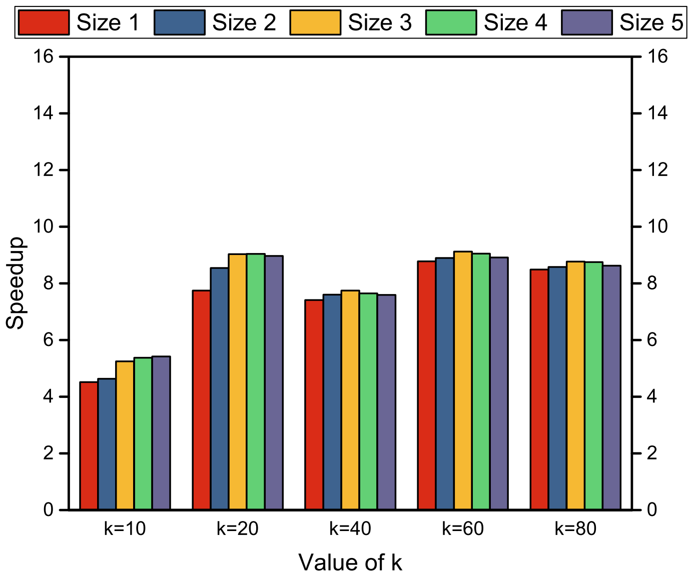
The experimental results illustrated in Fig. 7 also clearly indicate that: with the increase of the values of k, the speedups achieved by the parallel RBF-based spatial interpolation algorithm on the GPU become large. This behavior is quite reasonable since with the increase in the data size, the computational cost needed to perform the spatial interpolation is thus correspondingly increased. And in this case, the power of the GPU parallelization is highly exploited.

However, it is also should be noted that: after the value of k is larger than a specific threshold value, the advantage in parallelizing on the GPU is no longer obvious. This is because that: in the RBF-based spatial interpolation, it is needed to first form a coefficient matrix and then solve a system of linear equations to obtain the weights. The size of the coefficient matrix and the computational cost needed for the solving the system of linear equations are strongly determined by the value of k.

With the increase of the value k, the computational cost for calculating the prediction value of each interpolated point increase, and the power of the GPU parallelization is highly exploited. But, with the increase of the value k, the memory requirements for storing the coefficient matrix in the solving of the linear equations on the GPU global memory significantly increase. This would strongly decrease the computational efficiency.

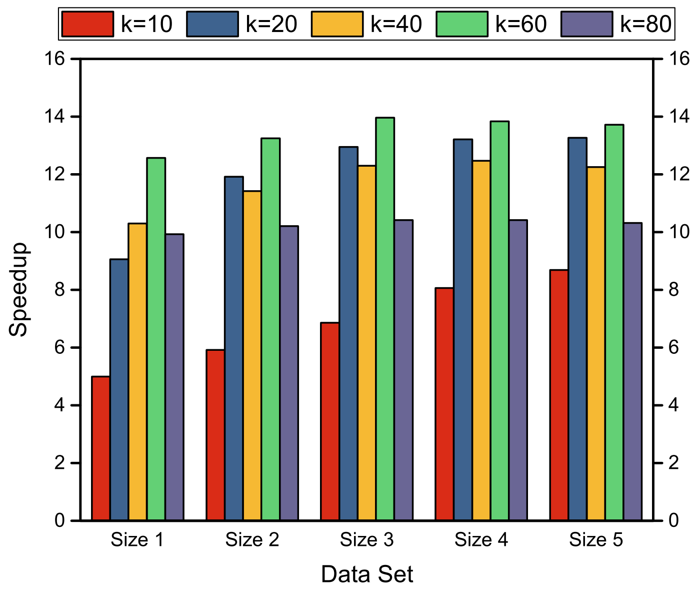


(a)

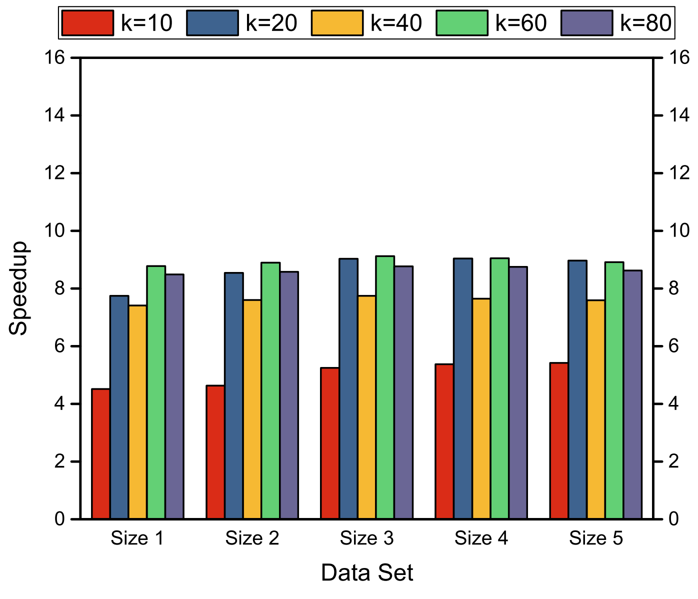


(b)

Fig. 6 Speedups achieved by the GPU-accelerated spatial interpolation algorithm when fixing the value of k. **a** On the PC No. 1. **b** On the PC NO.2



(a)



(b)

Fig. 7 Speedups achieved by the GPU-accelerated spatial interpolation algorithm when fixing the size of data sets. **a** On the PC NO.1. **b** On the PC NO.2

5.2 Impact of the Size of Data Set and the Value of *k* on the Interpolation Accuracy

In this subsection, we will discuss the impact of the size of data sets (i.e., the input data points and the interpolated points) and the different values of *k* on the interpolation accuracy of the proposed RBF-bases spatial interpolation algorithm for building explicit surfaces.

5.2.1 Impact of the Size of Data Set on the Interpolation Accuracy

To evaluate the impact the size of data set on the interpolation accuracy, we specifically re-organize the experimental results by fixing the value of *k* for five different sizes of data set; see Fig. 8.

The experimental results illustrated in Fig. 8 clearly indicate that: with the increase of the size of data sets, the interpolation error indicating by two metrics, NME and NRSME, always decreases. This behavior is quite reasonable since with the increase in the data size, i.e., the number of data points locating in the interpolation region increase, the density of data points (i.e., the number of data point locating in a unit square) increases.

If the density of data points is high, then the found *k* nearest data points for each interpolated point would be closely distributed. And in this case, the found *k* nearest data point would fit the underlying parametric surface quite well. When using those data points that well fit the underlying parametric surface to calculate the prediction value of an interpolated point, the interpolated point will be also well fit the underlying parametric surface; and thus, according to the definitions of the two metrics NME and NRSME, the interpolation accuracy is quite high.

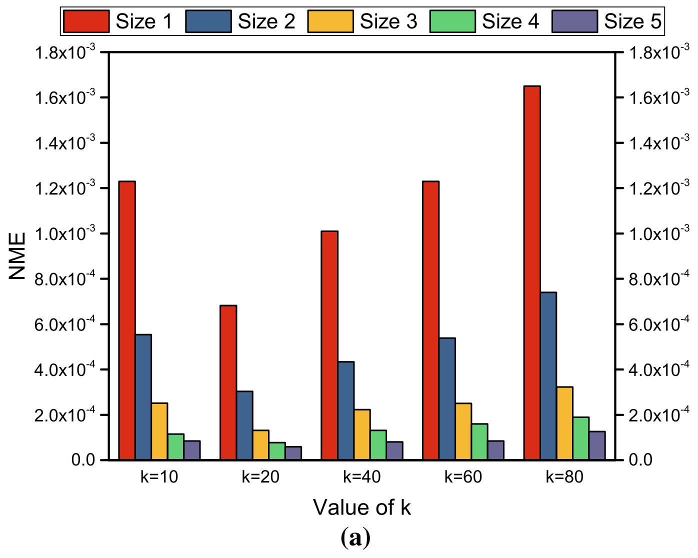
5.2.2 Impact of the Value of k on the Interpolation Accuracy

In our experiment tests, we have observed that: with the increase of the value of *k*, the interpolation accuracy in the same group of experimental tests become worse; see Fig. 9. The above behavior is interesting because in general the interpolation accuracy would be getting better when using a larger size of data points to calculate the prediction value.

We think it is mainly due to the following reason that leads to the above behavior.

The general concluding remark that the interpolation accuracy would be getting better when using a larger size of data points is highly dependent on the distribution pattern of the input data points and interpolated points. In this work, we create five groups of data points by randomly distributing them on a parametric surface (Eq. 11); and the interpolated points are also expected to be located on the parametric surface after predicting the *z* coordinates.

We have to carefully give notice that: the shape of the underlying parametric surface is not regular; see Fig. 5. Thus, the distribution pattern of all data points located on the underlying parametric surface is also not regular. When using a local set of data points to interpolate the prediction value of an interpolated point, the bigger is the size of the



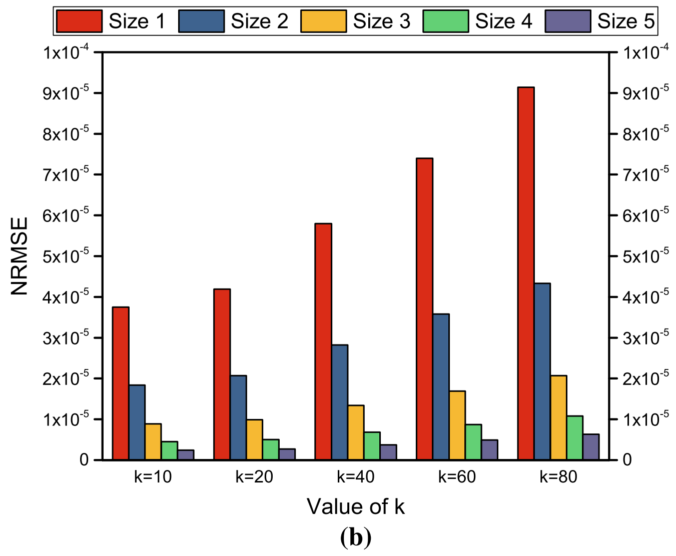
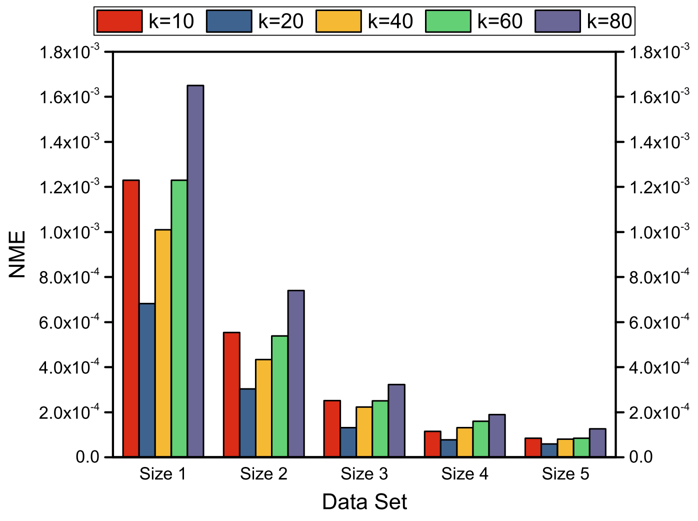
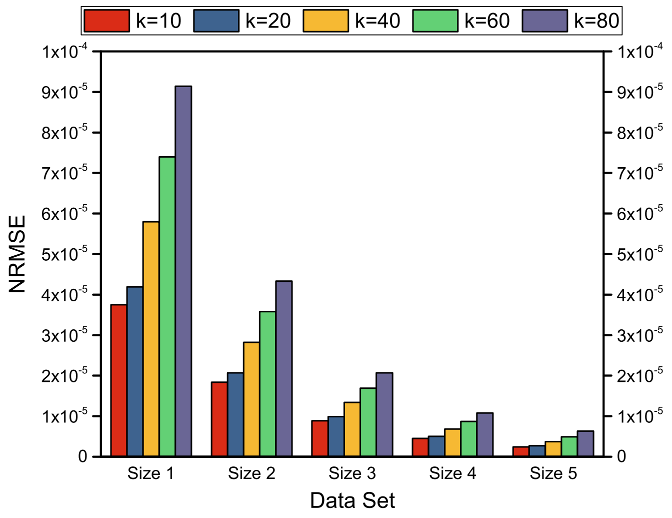


Fig. 8 Interpolation accuracy of the GPU-accelerated spatial interpolation algorithm when fixing the value of k. **a** NME. **b** NRMSE

used local set of data points, the more regularly the local set of data points distribute and the worse is the fitness of the local set of data points to the underlying parametric surface. In this case, the interpolation error arising in the predicting using a larger size of data points would correspondingly become larger.



(a)



(b)

Fig. 9 Interpolation accuracy of the GPU-accelerated spatial interpolation algorithm when fixing the size of data sets. **a** NME. **b** NRMSE

5.3 Key Factors that Limit the Efficiency of the GPU-Accelerated Spatial Interpolation Algorithm

5.3.1 The Memory Bottleneck in Developing GPU-Accelerated RBF Interpolation

In the GPU-accelerated spatial interpolation algorithm using RBFs, one of the performance bottlenecks is the memory allocation for storing the coefficient matrix in each

Table 15 Workload between the solving of linear equations and the rest part of calculation of the sequential version on the CPU on the PC NO.1 when the third group of data set is used

Value of k	Time of solving equations(/ms)	Time of the rest calculation (/ms)	Total time (/ms)	Percentage of the time of solving equations (%)	Percentage of the time of the rest calculation (%)
10	692	837	1529	45.3	54.7
20	2914	2226	5140	56.7	43.3
40	16953	7383	24336	69.7	30.3
60	49651	15448	65099	76.3	23.7
80	113319	26052	139371	81.3	18.7

thread. In the RBF-based spatial interpolation, it is needed first to form a coefficient matrix and then solve a system of linear equations to obtain the weights. The size of the coefficient matrix and the computational cost needed for the solving the system of linear equations are strongly determined by the value of k.

With the increase of the value k, the computational cost for calculating the prediction value of each interpolated point increase, and the power of the GPU parallelization is highly exploited. However, with the increase of the value k, the memory requirements for storing the coefficient matrix in the solving of the linear equations on the GPU global memory significantly increase. The global memory residing on a GPU is limited. All of the interpolated points cannot be mapped to the GPU to calculate the prediction values concurrently. To deal with the above problem, we have to first divide the interpolation points into several pieces on the CPU, and then transfer each piece of data points from the CPU to the GPU side. On the GPU, those interpolated points in each piece can be predicted in parallel.

Obviously, With the increase of the value k, the memory for storing the coefficient matrix becomes large, and the size of a piece of data point has to decrease. This further leads to the increase of the number of pieces and also the computational cost of transferring the data points from the CPU side to the GPU side.

In summary, due to the inherent feature in the GPU-accelerated spatial interpolation using RBFs, the memory for storing the coefficient matrix in each GPU thread is needed. And this memory allocation requirement strongly limits the computational efficiency.

5.3.2 Cost for Solving the System of Linear Equations

We have also found that: the solving of the system of linear equations in each thread is the major performance bottleneck that limits the computational efficiency of the parallel RBF-based spatial interpolation algorithm. To evaluate the computational cost on the solving of linear equations in the two implementations, we have specifically surveyed the time of solving equations, the time of the rest calculation, and the total time for the third group of data sets; see Tables 15 and 16.

Table 16 Workload between the solving of linear equations and the rest part of calculation of the parallel version on the GPU on the PC NO.1 when the third group of data set is used

Value of k	Time of solving equations (/ms)	Time of the rest calculation (/ms)	Total time (/ms)	Percentage of the time of solving equations (%)	Percentage of the time of the rest calculation (%)
10	167	56	223	74.9	25.1
20	229	168	397	57.7	42.3
40	1552	426	1978	78.5	21.5
60	3772	891	4663	80.9	19.1
80	11509	1869	13378	86.0	14.0

We have found that: the time cost on solving the equations significantly increase with the increase of the value of k. This is because that the number of linear equations is precisely the value of k. with the increase in the value of k, the size of the system of linear equations become larger. And correspondingly, the computational cost for solving the larger system of linear equations increases; see Fig. 10.

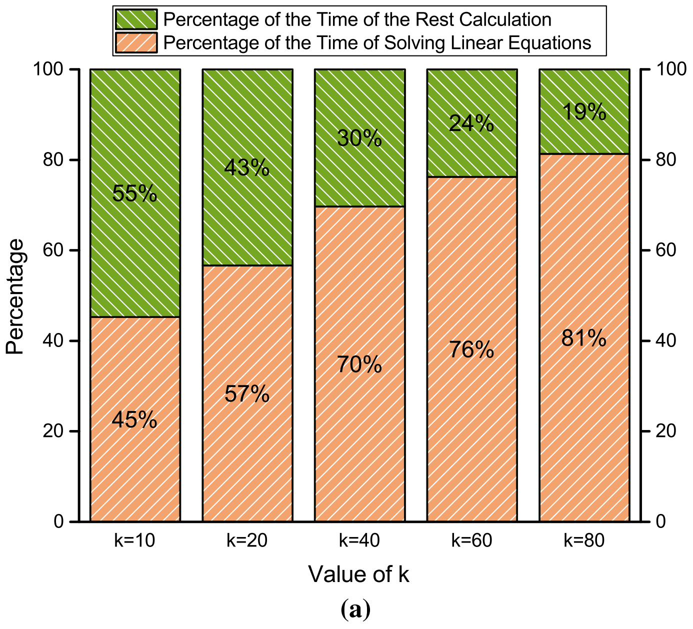
The computational efficiency of the parallel RBF-based spatial interpolation algorithm on many-core GPU is not much better than that of the parallel one on many-core CPU. The primary cause leading to the above behavior is the unsatisfied efficiency in solving the linear equations on the GPU.

The solving of a large system of linear equations can be well parallelized on the GPU. However, the solving of a relatively small system of linear equations in each GPU thread is not computationally efficient. This is because the solving of a system of linear equations using, for example, the Gaussian elimination method adopted in this work, is not suitable to be executed in a GPU thread. This is due to the fact that there are too many switch routines in the Gaussian elimination method.

In summary, in the parallel RBF-based spatial interpolation algorithm, it is needed to solve a system of linear equations using the Gaussian elimination method. However, the solving of linear equations is not suitable to be executed in a GPU thread, and thus strongly limits the computational efficiency of the parallel RBF-based spatial interpolation algorithm on the GPU.

5.4 Outlook and Future Work

In the presented GPU-accelerated spatial interpolation algorithm using RBFs for building explicit surfaces, there are two key ideas behind our method. The first idea is to employ a local set of data points rather than a global set of data points to calculate the prediction value of an interpolated point. The second is to use GS-RBFs rather than CS-RBFs to predict the desired z coordinate of an interpolation point. In the future, we plan to implement and evaluate the spatial interpolation using CS-RBFs to build explicit surfaces on the GPU.



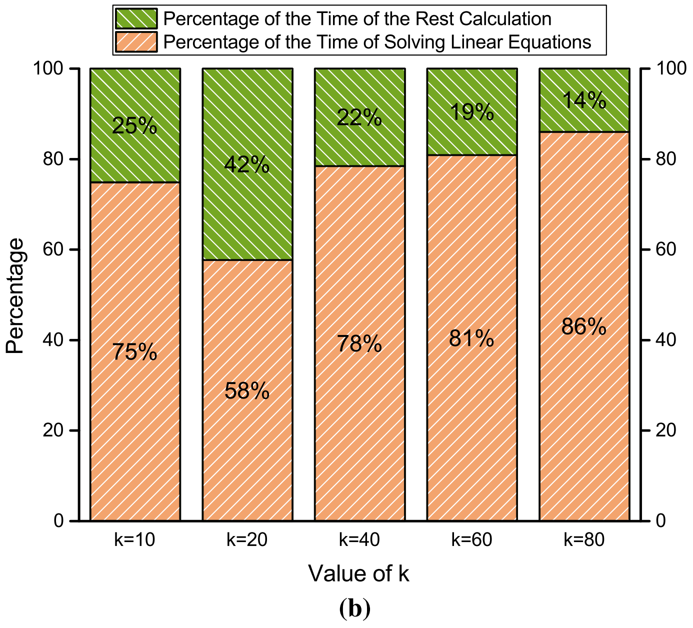


Fig. 10 Workload between the solving of linear equations and the rest part of calculation. **a** The sequential version on the CPU. **b** The parallel version on the GPU

6 Conclusions

In this paper, we have presented a parallel RBF-based spatial interpolation algorithm for building explicit surfaces by utilizing the power of modern GPUs. We have carried out five groups of experimental tests to evaluate the computational performance of the GPU-accelerated RBF interpolation algorithm by comparing it with the parallel implementation developed on multi-core CPUs. We have observed that: in most cases of our experimental tests, the presented parallel RBF-based spatial interpolation algorithm developed on many-core GPUs does not have any significant advantages over the parallel version implemented on multi-core CPUs in terms of computational efficiency. We have revealed that: (1) both the limited size of global memory and number of registers residing on the GPU are critical bottlenecks in developing an efficient GPU-accelerated spatial interpolation algorithm with RBFs; and (2) the need to solve a system of linear equations in each GPU thread to calculate the weights and prediction value of each interpolated point is another critical cause that results in the performance decrease of GPU-accelerated RBF interpolation algorithm.

Acknowledgements This research was jointly supported by the Natural Science Foundation of China (Grant Nos. 11602235 and 51674058), the China Postdoctoral Science Foundation (2015M571081), and the Fundamental Research Funds for China Central Universities (2652016105, 2652015065, and 2652017086).

References

1. Barlas, G.: Chapter 7–The Thrust Template Library, pp. 527–573. Morgan Kaufmann, Boston (2015). https://doi.org/10.1016/B978-0-12-417137-4.00007-1
2. Bell, N., Hoberock, J., Rodrigues, C.: Chapter 16—Thrust: A Productivity-Oriented Library for CUDA, pp. 339–358. Morgan Kaufmann, Boston (2013). https://doi.org/10.1016/B978-0-12-415992-1.00016-X
3. Carr, J.C., Beatson, R.K., Cherrie, J.B., Mitchell, T.J., Fright, W.R., McCallum, B.C., Evans, T.R., Acm, Acm: Reconstruction and representation of 3D objects with radial basis functions. In: Computer Graphics. Assoc Computing Machinery, New York, pp. 67–76. (2001). https://doi.org/10.1145/383259.383266
4. Chianese, A., Piccialli, F.: Smach: A framework for smart cultural heritage spaces. In: 10th International Conference on Signal-image Technology and Internet-Based Systems Sitis 2014, pp. 477–484. IEEE, 345 E 47TH ST, New York, NY 10017 USA (2014). https://doi.org/10.1109/SITIS.2014.16
5. Cuomo, S., De Michele, P., Piccialli, F.: 3D data denoising via nonlocal means filter by using parallel GPU strategies. Comput. Math. Methods Med. (2014). https://doi.org/10.1155/2014/523862
6. Cuomo, S., De Michele, P., Piccialli, F., Farina, R.: A smart GPU implementation of an elliptic kernel for an ocean global circulation model. Appl. Math. Sci. **7**, 3007–3021 (2013). https://doi.org/10.12988/ams.2013.13266
7. Cuomo, S., De Michele, P., Piccialli, F., Galletti, A., Jung, J.E.: IoT-based collaborative reputation system for associating visitors and artworks in a cultural scenario. Expert Syst. Appl. **79**, 101–111 (2017). https://doi.org/10.1016/j.eswa.2017.02.034
8. Cuomo, S., Galletti, A., Giunta, G., Marcellino, L.: Reconstruction of implicit curves and surfaces via RBF interpolation. Appl. Numer. Math. **116**, 157–171 (2017). https://doi.org/10.1016/j.apnum.2016.10.016
9. Cuomo, S., Galletti, A., Giunta, G., Starace, A.: Surface reconstruction from scattered point via RBF interpolation on GPU. In: Federated Conference on Computer Science and Information Systems, pp. 433–440. IEEE, New York (2013). https://doi.org/10.1145/383259.383266
10. D'Amore, L., Casaburi, D., Marcellino, L., Murli, A.: Numerical solution of diffusion models in biomedical imaging on multicore processors. Int. J. Biomed. Imaging **2011**, 680, 765 (2011). https://doi.org/10.1155/2011/680765

11. D'Amore, L., Marcellino, L., Mele, V., Romano, D.: Deconvolution of 3d fluorescence microscopy images using graphics processing units. In: Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 7203, pp. 690–699. Springer-Verlag Berlin, Heidelberger Platz 3, D-14197 Berlin, Germany (2012). https://doi.org/10.1007/978-3-642-31464-3_70
12. Franke, R.: Scattered data interpolation—tests of some methods. Math. Comput. **38**(157), 181–200 (1982). https://doi.org/10.2307/2007474
13. Hardy, R.L.: Multiquadric equations of topography and other irregular surfaces. J. Geophys. Res. **76**(8), 1905–1915 (1971). https://doi.org/10.1029/JB076i008p01905
14. Hillier, M.J., Schetselaar, E.M., de Kemp, E.A., Perron, G.: Three-dimensional modelling of geological surfaces using generalized interpolation with radial basis functions. Math. Geosci. **46**(8), 931–953 (2014). https://doi.org/10.1007/s11004-014-9540-3
15. Hoberock, J., Bell, N.: Thrust library (2017). http://thrust.github.io/
16. Huang, F., Bu, S.S., Tao, J., Tan, X.C.: OpenCL implementation of a parallel universal kriging algorithm for massive spatial data interpolation on heterogeneous systems. ISPRS Int. J. Geo-Inf. **5**(6), 22 (2016). https://doi.org/10.3390/ijgi5060096
17. Huang, F., Liu, D.S., Tan, X.C., Wang, J.A., Chen, Y.P., He, B.B.: Explorations of the implementation of a parallel IDW interpolation algorithm in a Linux cluster-based parallel GIS. Comput. Geosci. **37**(4), 426–434 (2011). https://doi.org/10.1016/j.cageo.2010.05.024
18. Izquierdo, D., de Silanes, M.C.L., Parra, M.C., Torrens, J.J.: CS-RBF interpolation of surfaces with vertical faults from scattered data. In: 4th International Conference on Approximation Methods and Numerical Modeling in Environment and Natural Resources—PART II, Mathematics and Computers in Simulation **102**, 11–23 (2014). https://doi.org/10.1016/j.matcom.2013.05.015
19. Karkouch, A., Mousannif, H., Al Moatassime, H., Noel, T.: Data quality in internet of things: a state-of-the-art survey. J. Netw. Comput. Appl. **73**, 57–81 (2016). https://doi.org/10.1016/j.jnca.2016.08.002
20. Lin, Y., Chen, C., Song, M., Liu, Z.: Dual-RBF based surface reconstruction. Vis. Comput. **25**(5), 599–607 (2009). https://doi.org/10.1007/s00371-009-0349-x
21. Lu, G., Ren, L., Kolagunda, A., Wang, X., Turkbey, I.B., Choyke, P.L., Kambhamettu, C.: Representing 3D shapes based on implicit surface functions learned from RBF neural networks. J. Vis. Commun. Image Represent. **40**, 852–860 (2016). https://doi.org/10.1016/j.jvcir.2016.08.014
22. Luo, S.H., Wang, J.X., Wu, S.L., Xiao, K.: Chaos RBF dynamics surface control of brushless DC motor with time delay based on tangent barrier lyapunov function. Nonlinear Dyn. **78**(2), 1193–1204 (2014). https://doi.org/10.1007/s11071-014-1507-x
23. Macêdo, I., Gois, J.a.P., Velho, L.: Hermite interpolation of implicit surfaces with radial basis functions. In: Proceedings of the 2009 XXII Brazilian Symposium on Computer Graphics and Image Processing, SIBGRAPI '09, pp. 1–8. IEEE Computer Society, Washington, DC, USA (2009). https://doi.org/10.1109/SIBGRAPI.2009.11
24. Mallet, J.L.: Discrete smooth interpolation in geometric modeling. Comput. Aided Des. **24**(4), 178–191 (1992). https://doi.org/10.1016/0010-4485(92)90054-e
25. Mallet, J.L.: Discrete modeling for natural objects. Math. Geol. **29**(2), 199–219 (1997). https://doi.org/10.1007/bf02769628
26. Matheron, G.: Principles of geostatistics. Econ. Geol. **58**(8), 1246–1266 (1963). https://doi.org/10.2113/gsecongeo.58.8.1246
27. Mei, G., Tipper, J.C., Xu, N.: A generic paradigm for accelerating Laplacian-based mesh smoothing on the GPU. Arab. J. Sci. Eng. **39**(11), 7907–7921 (2014). https://doi.org/10.1007/s13369-014-1406-y
28. Mei, G., Xu, L., Xu, N.: Accelerating adaptive inverse distance weighting interpolation algorithm on a graphics processing unit. R. Soc. Open Sci. (2017). https://doi.org/10.1098/rsos.170436
29. Mei, G., Xu, N.X., Xu, L.L.: Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search. Springerplus **5**, 22 (2016). https://doi.org/10.1186/s40064-016-3035-2
30. Mirzaei, D.: Analysis of moving least squares approximation revisited. J. Comput. Appl. Math. **282**, 237–250 (2015). https://doi.org/10.1016/j.cam.2015.01.007
31. Piccialli, F., Cuomo, S., De Michele, P.: A regularized MRI image reconstruction based on hessian penalty term on CPU/GPU systems. In: 2013 International Conference on Computational Science, Procedia Computer Science, vol. 18, pp. 2643–2646. Elsevier Science BV, Sara Burgerhartstraat 25, Po Box 211, 1000 Ae Amsterdam, Netherlands (2013). https://doi.org/10.1016/j.procs.2013.06.001

32. Shankar, V., Wright, G.B., Kirby, R.M., Fogelson, A.L.: A radial basis function (RBF)-finite difference (FD) method for diffusion and reaction–diffusion equations on surfaces. J. Sci. Comput. **63**(3), 745–768 (2015). https://doi.org/10.1007/s10915-014-9914-1
33. Shepard, D.: A two-dimensional interpolation function for irregularly-spaced data (1968). https://doi.org/10.1145/800186.810616
34. Wang, J.G., Liu, G.R.: On the optimal shape parameters of radial basis functions used for 2-D meshless methods. Comput. Methods Appl. Mech. Eng. **191**(23–24), 2611–2630 (2002). https://doi.org/10.1016/s0045-7825(01)00419-4
35. Wang, J.G., Liu, G.R.: A point interpolation meshless method based on radial basis functions. Int. J. Numer. Methods Eng. **54**(11), 1623–1648 (2002). https://doi.org/10.1002/nme.489
36. Wang, Q., Pan, Z., Bu, J., Chen, C.: Parallel RBF-based reconstruction from contour dataset. In: 2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics, pp. 82–85 (2007). https://doi.org/10.1109/CADCG.2007.4407860
37. Yang, R., Er, P.V., Wang, Z., Tan, K.K.: An RBF neural network approach towards precision motion system with selective sensor fusion. Neurocomputing **199**, 31–39 (2016). https://doi.org/10.1016/j.neucom.2016.01.093
38. Yokota, R., Barba, L.A., Knepley, M.G.: PetRBF—a parallel $o(n)$ algorithm for radial basis function interpolation with gaussians. Comput. Methods Appl. Mech. Eng. **199**(25–28), 1793–1804 (2010). https://doi.org/10.1016/j.cma.2010.02.008