Computers & Geosciences 105 (2017) 129–138

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Research paper

GPU based contouring method on grid DEM data

Liheng Tan*, Gang Wan, Feng Li, Xiaohui Chen, Wenlong Du

Institute for Geographic Spatial Information, The PLA Information Engineering University, Zhengzhou, Henan 450000, China

ARTICLE INFO

Keywords:
Contour
Grid DEM
GPGPU
Geometry shader
Spatial interpolation

ABSTRACT

This paper presents a novel method to generate contour lines from grid DEM data based on the programmable GPU pipeline. The previous contouring approaches often use CPU to construct a finite element mesh from the raw DEM data, and then extract contour segments from the elements. They also need a tracing or sorting strategy to generate the final continuous contours. These approaches can be heavily CPU-costing and time-consuming. Meanwhile the generated contours would be unsmooth if the raw data is sparsely distributed. Unlike the CPU approaches, we employ the GPU's vertex shader to generate a triangular mesh with arbitrary user-defined density, in which the height of each vertex is calculated through a third-order Cardinal spline function. Then in the same frame, segments are extracted from the triangles by the geometry shader, and translated to the CPU-side with an internal order in the GPU's transform feedback stage. Finally we propose a "Grid Sorting" algorithm to achieve the continuous contour lines by travelling the segments only once. Our method makes use of multiple stages of GPU pipeline for computation, which can generate smooth contour lines, and is significantly faster than the previous CPU approaches. The algorithm can be easily implemented with OpenGL 3.3 API or higher on consumer-level PCs.

1. Introduction

Contours, grid DEM and TIN DEM are the three most important forms of terrain expression (Wang and Wu, 2006), each of which has its own characteristics and usages. The conversion between them is one of the frequent studies in the research field of GIS. Among the three, contours are measurable, and can be perceived directly. Therefore they play an important role in cartography, map reading and terrain analysis.

In recent years, the production of DEM data has enhanced greatly in quantity, quality, range and other aspects (Jaara and Lecordix, 2011). The majority of them are released as grid DEM, such as SRTM, GDEM, DLR, etc. As a result, the demand of generating contours from DEMs is increasingly urgent, as an important function in computer-aided cartography (Li and Zhu, 2003). Particularly, extracting contours from grid DEM has become a focus of study.

Contour is a kind of isoline, the study about which began since the 1960s (Barcha and Reese, 1964). After decades of research and development, various ideas and methods have been put forward, which can be analyzed and summarized from different aspects.

The contouring algorithms can be classified into three categories according to the final results: ①Vector contour. These methods usually use the finite element mesh (FEM) to represent the terrain surface. The attribute values (e.g., elevation) are acquired for all vertexes. Then contour line segments are extracted from the elements according to linear interpolation. This is the typical process used by most of the contouring algorithms. ②Raster contour image. The results of these approaches are raster images with specific resolutions, on which the contours are rendered. For each pixel of the image, its color is calculated according to the attribute value in its actual spatial position (Wang et al., 2001; Paul, 1988; Gul and Khan, 2010; Schlei, 2009). Chen et al. (2010) implement this method on GPU: Render the triangular mesh of terrain through the graphics pipeline, and compare the elevation values of the contours and each pixel in the pixel shader to calculate the pixel's color and opacity value, which could achieve a better blending effect. Although GPU can accelerate the calculation by its pixel-wise parallel mechanism, however, the principle it follows is still linear interpolation of the three vertices of a triangle. Therefore it can't obtain a fine smooth result if the mesh is not dense enough. On the other hand, the final results of these methods are raster images, rather than vector contours, which can't totally meet the requirements of GIS applications. ③Contour functions. Zhao (2003) also uses regular grid data. He employs a ternary cubic equation to fit each grid cell. So for a certain height value, the smooth contour line in the cell can be solved as a binary cubic equation, which can be fully presented and stored with only 8 coefficients. But obtaining these coefficients requires to resolve a set of 10-elements linear equations, which is computationally expensive. Besides, the form of parameter equation can't

* Corresponding author.
E-mail address: star-force@qq.com (L. Tan).

http://dx.doi.org/10.1016/j.cageo.2017.05.007
Received 11 October 2016; Received in revised form 15 May 2017; Accepted 16 May 2017
Available online 18 May 2017

describe the geometry directly, so its usage is restricted. Similarly, Bryan et al. (2005) employ the radial basis function (RBF) to fit all the data sampling points for each cell, and define the contour lines by implicit functions. But this way can neither get a global differentiable function, nor can it make the usage of the result more extensive.

The first kind of method - vector contour – is most commonly used, and can be further subdivided into two categories by the basic elements of the FEM: grids or triangles. ① Carsten (2003) describes the contour generation algorithm based on a grid FEM, namely Marching Squares (MS). This algorithm summarizes all 16 possible spatial relationships between a single contour segment and a grid cell. The certain type of relationship can be quickly determined by comparing the height values of the four vertexes and the contour. Then the segment can be directly computed. Similar works are presented respectively with their own data structures and processes (You, 1989; Xie and Tian, 1995; Wang et al., 2007a, 2007b; Sun et al., 2000; Li, 2010; Chester et al., 1968; Hiremath and Kodge, 2010). ② Triangular mesh based methods are also proposed (Cheng et al., 1998; Miao, 2004; Rui et al., 2011; Shao et al., 2014; Goldin and Gao, 2006; Rognant et al., 2001; Watson, 1982), which can reduce the quantity of types of spatial relationships to 8. Thus segment extraction can be simplified. However, the tracing or sorting processes may become more complicated.

The vector-contour methods can also be subdivided into two major classes by their workflows, including contour sorting and contour tracing (Watson, 1993). ① The sorting methods first try to find contour segments from all of the FEM cells respectively, and will obtain a set of contour segments with random order. Then these segments are sorted and organized into strings or loops to obtain the full contour lines (Nickerson et al., 1999). The advantage of these methods is that the segments extracting stage is quite simple and highly parallel, so it is easy to be accelerated (Xie, 2012). The sorting stage is the focus of researches. Norman et al. (2000) sort the segments from different directions successively, and in some cases it can be more efficient than the tracing method. ② The tracing methods generate one complete contour line each time, segment by segment. They usually begin by finding a useful FEM cell. After obtaining a starting contour segment in the cell, they try to find the continuous segments from the neighbor cells (Buys et al., 1991; Yates, 1987; Lodwick and Whittle, 1970; Kok and Begin, 1981). This process continues until the contour line reaches the boundary or gets closure. Theoretically, the tracing methods can find all contours by traversing the cells in just one pass, so they can be more efficient. Also there is a vast literature which aims to find the starting cell more efficiently (Van and Marc, 1996, 2006; , 2007a, 2007b; Zhao et al., 2014), with techniques such as interval tree, binary tree, quick sorting, bucket index, etc. However, these preprocessing steps need additional traversals of the cells, thus the overall computational cost may be even more expensive.

The contour generation algorithms also need to meet the problem of smoothness (Schmieder and Huber, 2000). This problem comes up when the original data is sparsely distributed, so will the FEM be sparse too. For each cell of the FEM, the possible segment is a straight line according to linear interpolation. So the whole contour line will be quite jagged, which is neither beautiful nor realistic. For fitting the raw contour polylines into smoothed results, cubic Bezier curve, cubic B-spline, parabola, oblique axis parabola and other methods are widely discussed (Sun et al., 2000; Miao, 2004; Wang, 2006). However, these curve-based smoothing approaches ignore the planar topological relations between lines, so the results may be self-intersected (Fig. 1) (Riegler et al., 2006). Some others employ the analytic function methods (Zhang et al., 2001; Rui et al., 2011), and treat each FEM element (grid or triangle) as a function surface, such as bivariate quadratic function, bivariate cubic function etc. Then the original elements are subdivided to achieve a denser FEM, from which the smoother contour lines could be obtained. Zhang (1991) uses the similar idea, except that he solves the contour's function expression directly from the bivariate cubic cell surface, then discretizes it to

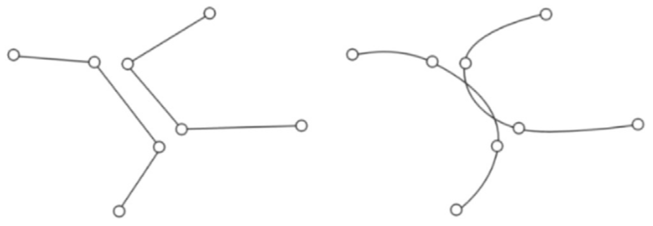


Fig. 1. Self-intersection of the smoothed contour lines.

obtain the smooth contour's points list. However, these methods can only ensure the final contours remain smooth in each FEM cell. The lines are still not globally differentiable. So Zhang and Liang (1997) first subdivide FEM cells to achieve local smoothness, then use the cubic spline for global smoothness. But still, self-intersection may occur in some strict conditions. Besides, Peters et al. (2014), Kolingerová et al. (2009) concentrate on altering the height values or triangulation of the original FEM to eliminate the aliasing and roughness. However, these approaches have little effect for the sparse condition.

In this paper, we present a novel contour sorting method on Grid DEM data, which can be easily integrated into 3D GIS system. The segments extracting stage is significantly accelerated through the programmable GPU pipeline, by using vertex shader, geometry shader and transform feedback. Based on the GPU's internal feedback order, a highly efficient “Grid Sorting” algorithm is also proposed to sort the segments and generate the final complete contours. Moreover, we employ a third-order Cardinal spline to achieve a C^3continuous description of the entire terrain by GPU, and also use a user-defined arbitrarily dense triangular FEM as the foundation of contouring. Thus the problem of global smoothness can be solved exactly. Our method makes use of multiple stages of modern GPU pipeline, and gets improvements on both effect and efficiency over the previous approaches.

2. GPU spatial interpolation

2.1. Use the DEM data properly

Digital Elevation Model (DEM) is a 3D digital representation of terrain surface. However, the DEM data is usually a set of discrete sampling points, meanwhile most of the actual terrain surface is continuous. Therefore we need to define the continuous function of the terrain surface, as $h = f(x, y)$, which can not only fit all of the sampling points, but also can determine the height value for arbitrary position in the domain. Contour generation must be concerned with function h. The contouring results may vary a lot when different functions are used, even though the sampling points are the same (Grain, 1970).

For grid DEM data, traditional contour generation algorithms usually focus on the discrete sampling points only (Fig. 2.a). They often directly connect the sampling points to construct a FEM for contour generation, which means, they think the terrain surface appears as shown in Fig. 2.c. Because the raw data based triangular network interpolates the elevation values on its inner spatial locations by linear interpolation (barycentric interpolation) potentially.

This description cannot be globally smooth, and finally appears as a 2D piecewise function. In each triangle, the function is C^1continuous as a linear plane. However it could not be differentiable when comes to the edges of the triangles. As a result, this description can’t fit the real terrain surface well if the sampling points are relatively sparse (Fig. 2.a), nor can it lead to the generation of smooth contours. Similar descriptions include the Voronoi polygon partition (Fig. 2.b), namely nearest interpolation, which is neither continuous nor differentiable; and bilinear interpolation (Fig. 2.d), which can achieve C^1

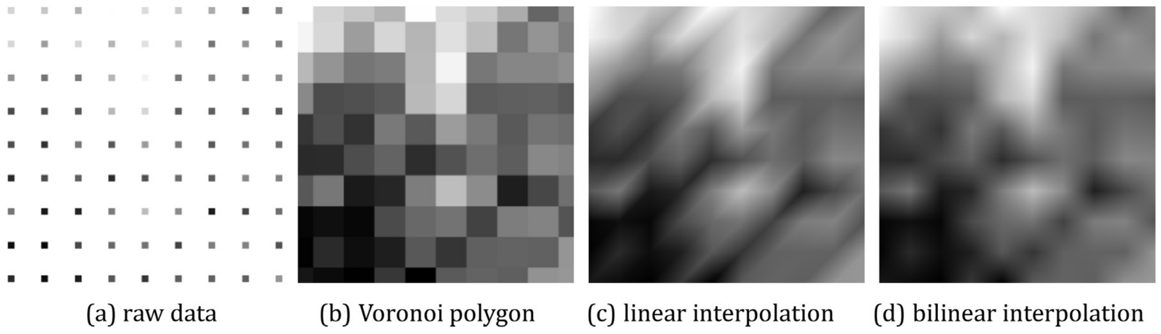


Fig. 2. Different mathematical descriptions on the same sampling points.

continuity within a square area, but is still not differentiable through the edges. These methods could not get a global continuous and differentiable description about the ground surface either.

2.2. GPU cardinal spline interpolation for terrain description

As previously mentioned, to generate high quality, smooth contour lines, it is necessary to obtain a global, at least C^2continuous description from the raw sampling points. In the field of spatial interpolation, many relevant researches have been proposed. The commonly used interpolation methods can be classified from lots of aspects. According to the operation strategies, there are global interpolation, piecewise interpolation, statistics based interpolation and inverse distance weighting (Franke, 1982; Nielson, 1993). From the perspective of interpolating function, there are polynomial interpolation, Lagrange interpolation, Newton interpolation, spline interpolation and the radial basis function interpolation, etc (Schoenberg, 1969, Schoenberg and Sharma, 1973). From the number of variables used each time, there are multivariate interpolation (Akima, 1978) and multi-dimensional univariate interpolation (Lee et al., 1997).

When the amount of DEM data is very large, the global interpolation methods generally employ the high order polynomial function to fit all of the sampling points. However, this function is hard to solve, and a serious Runge's phenomenon may occur. So in this paper, we prefer to use the piecewise interpolation strategy, which is parallelizable, and is suitable to implement on GPU. In terms of selecting the interpolation function, we notice that the spline functions only need a small amount of sampling points, and can ensure that the whole terrain has C^3 continuity. The most commonly used spline functions are Thin-Plate spline, B-spline, Hermite spline, Bezier-Bernstein spline, Cardinal spline, Cone spline, Box spline, etc. Wherein the Cone spline and Box spline are multivariate functions, which can be used to interpolate the 2D surface directly (Chui, 1988); the others are univariate splines, which can also describe 2D continuous surface if we use it twice and for one dimension each time.

Spline interpolation is more often used in computer graphics and computer-aided design; e.g. high order texture filters are proposed to achieve a better rendering quality (Bjorke, 2004; Christian and Markus, 2005). In this paper, we argue that the Cardinal spline is very suitable for describing terrain. On the one hand, it employs cubic polynomial functions as its basis functions, thus it can keep C^3 continuity with simple calculation; on the other, a Cardinal spline can change its tightness easily by adjusting the tension parameter. So it has better adaptability for different kinds of terrains. In conclusion, we use the 2D piecewise Cardinal spline interpolation to fit the raw data with a smooth surface.

The general form of the cubic Cardinal spline is as follows (Lalescu, 2009):

$$f_i(u) = [u^3 u^2 u 1] \bullet \tau \bullet \begin{bmatrix} -1 & 2/\tau-1 & -2/\tau+1 & 1 \\ 2 & -3/\tau+1 & 3/\tau-2 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1/\tau & 0 & 0 \end{bmatrix} \bullet \begin{bmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{bmatrix} \tag{1}$$

where p_{i-1}, p_i, p_{i+1}, p_{i+2} are four control points. $f_i(u)$ is the piecewise Cardinal spline function which is used to interpolate from p_i to p_{i+1}. Here $u \in [0,1]$, and $f_i(0) = p_i$ and $f_i(1) = p_{i+1}$. The curve would extend and still keep global C^3 continuity along with the increase of i. τ is the tension parameter, which should range from 0 to 1 for DEM interpolation. The smaller τ is, the tighter the spline will be, and vice versa (Fig. 3). So theoretically when it comes to different kinds of terrains, whether rugged or flat, we can choose the appropriate τ value for spatial interpolation. However, there are few researches on DEM interpolation with Cardinal spline in the present. The quantitative criterion for choosing the appropriate τ values for different types of terrains is still uncertain. This is a little beyond the scope of this paper and may take a lot of discussion, so we intend to fulfill this work in our following study.

This function is univariate, so we use it twice for 2D DEM interpolation, first vertically (along with the longitude line), then horizontally (along with the latitude line). Then we can get a smooth terrain description. As shown in Fig. 4, for any point $p(x, y)$ in the data field, we can calculate its height value according to the 16 neighbor sampling points. In addition, we implement our method on programmable GPU in vertex shader stage with the feature of vertex texture fetch, which is useful for the FEM generation later. Owing to the highly parallel architecture of GPU, the interpolation process can be extremely fast.

Our approach is as follows:

① Store the grid DEM data (height map) as a rectangle texture, with 32-bits float precision, nearest filter, and alpha channel only. Use it as an input into the graphics pipeline.
② The horizontal line that pass $p(x, y)$ must intersect the raw data grid with 4 points, 2 for each side. Denote them as $p_i(x_i, y_i)$, $i \in \{1,2,3,4\}$. Then solve the 2D plane coordinates for them.
③ For each point $p_i(x_i, y_i)$, use the 4 raw DEM data points around it in the vertical direction as sampling points, also 2 for each side.

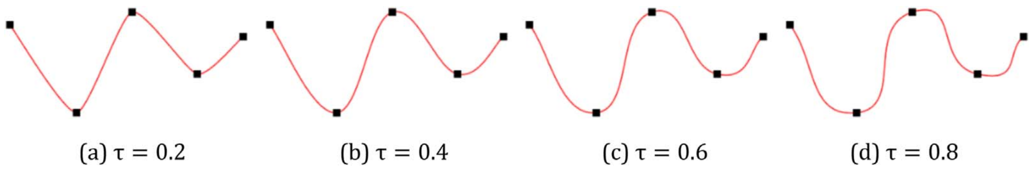
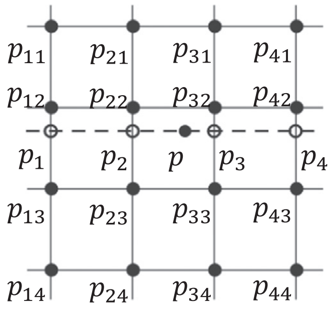


Fig. 3. The shapes of Cardinal spline with different τ values.



p_{11} p_{21} p_{31} p_{41}
p_{12} p_{22} p_{32} p_{42}
p_1 p_2 p p_3 p_4
p_{13} p_{23} p_{33} p_{43}
p_{14} p_{24} p_{34} p_{44}

Fig. 4. 2D Cardinal spline interpolation.

Fetch the DEM texture, and get their 3D coordinates as $p_{ij}(x_{ij}, y_{ij}, z_{ij}), j \in \{1,2,3,4\}$. Normalize the vertical coordinate y_i to u_i with the following equation:

$$u_i = (y_i - y_{i2})/(y_{i3} - y_{i2}) \tag{2}$$

Then substitute u_i and $p_{ij}\,(j \in \{1,2,3,4\})$ into Eq. (1), so we can get the elevation value on point p_i, in form of its 3D coordinate $p_i(x_i, y_i, z_i)$.

④ Similarly, using $p_i\,(\mathrm{i} \in \{1,2,3,4\})$ as sampling points, and here:

$$u = (x - x_2)/(x_3 - x_2) \tag{3}$$

Again substitute them into Eq. (1), then we can get the elevation value on point p.

Here are the GLSL codes for the interpolation in vertex shader:

```
//Cardinal Spline coefficient matrix
uniform mat4 SplineMatrix
// DEM height map texture
uniform sampler2DRect _tex
// the texture coordinates of any point p(x,y)
vec2 uv
float caculateSpline(float u,vec4 _heightvalues)
{
   vec4 l_vU = vec4(u*u*u,u*u,u,1.0);
   return dot(l_vU,SplineMatrix*_heightvalues);
}
float TextureSampleAlphaSpline(sampler2DRect _tex,vec2 uv)
{
   vec2 frac_xy = uv − floor(uv);
   vec2 xy = floor(uv);
   vec4 l_values;
   l_values=vec4(texture(_tex,xy+vec2(−1.0,−1.0)),texture(_tex,xy
     +vec2(−1.0,0.0)),/
   texture(_tex,xy+vec2(−1.0,1.0)),texture(_tex,xy
     +vec2(−1.0,2.0)));
   float col0 = caculateSpline(frac_xy.y,l_values);
   l_values = vec4(texture(_tex,xy+vec2(0.0,−1.0)),texture(_tex,xy
     +vec2(0.0,0.0)),/
   texture(_tex,xy+vec2(0.0,1.0)),texture(_tex,xy+vec2(0.0,2.0)));
   float col1 = caculateSpline(frac_xy.y,l_values);
   l_values = vec4(texture(_tex, xy
     +vec2(1.0,−1.0)),texture(_tex,xy+vec2(1.0,0.0)),/
   texture(_tex,xy+vec2(1.0,1.0)),texture(_tex,xy+vec2(1.0,2.0)));
   float col2 = caculateSpline(frac_xy.y,l_values);
   l_values = vec4(texture(_tex,xy+vec2(2.0,−1.0)),texture(_tex,xy
     +vec2(2.0,0.0)),/
   texture(_tex,xy+vec2(2.0,1.0)),texture(_tex,xy+vec2(2.0,2.0)));
   float col3 = caculateSpline(frac_xy.y,l_values);
   return caculateSpline(frac_xy.x,vec4(col0,col1,col2,col3));
}
```

The results are shown in Fig. 5. Obviously our method can get a global high-order continuous terrain description, which goes from rough to flat along with the increase of τ.

To implement this method in GPU, the key point is to map the real geographic coordinates to the texture coordinates correctly, then we can access the corresponding texel from the raw DEM data for any position. The matrix from Eq. (1) can be calculated in CPU-side with a certain τ value, and then transmit into GPU pipeline as a 16-floats uniform parameter. The whole calculation process is quite simple, only needs 16 times of texture fetching, and several times of matrix-vector multiplications. For modern GPU devices, the time cost for interpolating millions of points is sub-millisecond, which is negligible for most cases. It can be directly incorporated into the contouring process.

2.3. FEM construction

The contour sorting algorithm still needs a FEM to perform on. However, benefiting from the previous C^3 continuous surface we achieved, the resolution of the FEM doesn't need to be restricted to the raw DEM data, as the previous approaches (Fig. 6.a). Instead, we use a user-defined triangulated FEM with arbitrary mesh density for contouring (Fig. 6.b). For each vertex of the mesh, the method introduced in 2.2 is used to obtain the elevation value.

In fact, a spline function is a numeric function that is piecewise defined by polynomial functions, which possesses a high degree of smoothness at any position in its domain. In step 2.2, we use cubic Cardinal spline twice to form the interpolation surface, first vertically, then horizontally. So finally this surface is a piecewise-defined bivariate cubic polynomial function, which is C^3 continuous in its domain. Assuming the surface function is $\mathrm{z} = \mathrm{f}(\mathrm{x}, \mathrm{y})$, then the functions of contours are $\mathrm{f}(\mathrm{x}, \mathrm{y}) = \mathrm{h_i}(\mathrm{i} = 1, 2, 3\ldots)$, where $\mathrm{h_i}$ are the height values of contours. Now the gradient function of the contour line can be solved as $\nabla\mathrm{f} = \mathrm{f_x i} + \mathrm{f_y j}$. Since $\mathrm{f}(\mathrm{x}, \mathrm{y})$ is a three order polynomial function, then $\mathrm{f_x}$, $\mathrm{f_y}$ and $\nabla\mathrm{f}$ will be C^2 continuous, which means the contour functions will be smooth.

Meanwhile, the FEM surface is a continuous piecewise bivariate linear function, assuming it is $z = F(x, y)$. For any position $p(x_p, y_p)$ in the domain, it will be in one triangle, marked as $p_0p_1p_2$. Assume the density of FEM is ρ, and the length of the longest edge of the triangle is φ. Clearly, if $\rho\to\infty$, then $\varphi\to 0$, so $p_0, p_1, p_2 \to p$. And $z = F(x, y)$ and $z = f(x, y)$ are both continuous functions, so $F(x_p, y_p) = F(x_0, y_0) = f(x_0, y_0) = f(x_p, y_p)$, which means:

$$\lim_{\rho\to\infty} F(x, y) = f(x, y)$$

So when the FEM is dense enough and the coordinates of its vertexes have been calculated, its surface will tend to be similar with the piecewise spline function. Thus, the contours extracted will be smooth at visual scale. And the smoothness of contour lines is due to the smooth surface, so it will not get self-intersection at all.

In our opinion, the reasonable resolution of the FEM (R_{FEM}) is related to both the resolution of the raw DEM data (R_{dem}) and the resolution of the view (R_{view}). R_{FEM} should be greater than R_{dem}, otherwise the FEM can't cover the fundamental information of the raw data. Meanwhile R_{FEM} should be greater than R_{view} too, because it is also important to obtain a fine visual effect. So we can conclude:

$$R_{FEM} \geq \max(R_{dem}, R_{view}) \tag{4}$$

R_{view} is correlated with the view size. Here an empirical value can be recommended as:

$$R_{view} \approx (ViewWidth/0.\,2cm) \times (ViewHeight/0.\,2cm) \tag{5}$$

For example, for the DEM data shown in Fig. 6.a, its R_{dem} is 10×10. When it is rendered on screen with the size of $20cm \times 20cm$, the R_{view} will be about 100×100 according to Eq. (5), which is greater than R_{dem}. So here R_{FEM} should be greater than R_{view}. With the FEM of this

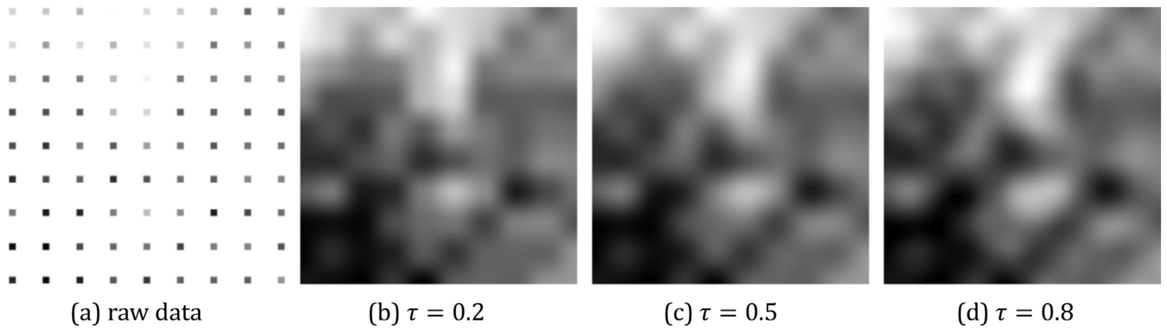


Fig. 5. Cardinal spline interpolation results with different τ values.

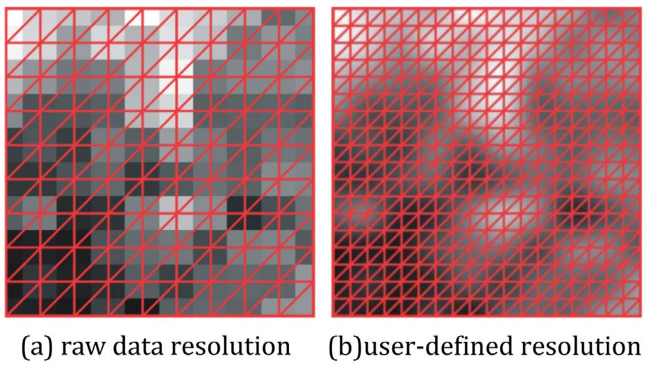


Fig. 6. FEMs of different resolutions. (a) raw data resolution (b)user-defined resolution.

resolution, we can achieve very smooth results with full of details.

3. GPU contouring

3.1. Overview

GPU is originally designed for acceleration of rendering 3D graphics. In recent years, with the rapid development of its data processing ability and programmable pipeline architecture, GPGPU (general-purpose computing on graphics processing units) has been proposed and widely discussed (Qian et al., 2014). Some very popular GPU-based parallel computing platforms also appeared, such as CUDA and OpenCL. Still, when it is needed to combine with the existing graphic system, like a 3D GIS platform, we can directly use the standard graphics pipeline to perform parallel computing. Through this way, the flow of the system can be simpler. Especially for contour generation, we just need to modify some shader stages of the graphics pipeline, and the contours can be generated and visualized in one single rendering pass. Besides, it does not need to grow dependent on third-party libraries, runtimes or drivers.

As shown in Fig. 7, the modern graphics pipeline consists of multiple stages, including vertex shader, fragment shader, etc. Especially the geometry shader, which is introduced with OpenGL 3.2 in 2009, has the ability to flexibly increase, decrease or change the geometry primitives. This is the key feature for contour segments extraction. Meanwhile another vertex post-processing mode – transform feedback – is provided, by which the altered primitives could be transferred back to the CPU-side for further usage.

Benefiting from these new functions and the inherent parallel computing mechanism, efficient GPU based contouring method could finally be realized.

The basic steps of our approach are as follows:

① Generate a 2D triangular FEM in CPU-side, using the method from Section 2.3.
② Pass the 2D FEM and the DEM texture into the vertex shader. In this stage, obtain the elevation value and get the 3D coordinate for each vertex, as shown in Section 2.2.
③ Then in geometry shader, change the triangles which have contours across them into contour line segments, then transmit them back to CPU using the transform feedback feature. Also transmit some auxiliary information back for efficient sorting.
④ Back in CPU-side, adopt the “Grid Sorting” method to make the segments into sorted strings or loops. So we can finally complete the generation of contours.

A flow chart of the algorithm is shown in Fig. 8. The first two steps have been explained in the chapter 2. We will now present the rest of them in details.

3.2. Segments extraction and feedback

Many papers have been written on the extraction of a possible segment from a triangle (Zhang and Liang, 1997; Cheng et al., 1998; Miao, 2004; Zhao et al., 2014; Hou et al., 2008). As shown in Fig. 9, the three vertexes of the triangle could be denoted by $p_i(x_i, y_i, z_i), i \in \{0,1,2\}$, and z_i is the elevation value. If the height of the contour is h, then a necessary and sufficient condition for the existence of a contour point in the edge $\overline{p_m p_n}$ $(m, n \in \{0,1,2\}, m \neq n)$ is:

$$(z_m - h)\bullet(z_n - h) < 0 \tag{6}$$

When the point exists, let's set it p, the formula for calculating its coordinate is:

$$p = p_m + (p_n - p_m)\bullet(h - z_m)/(z_n - z_m) \tag{7}$$

So for each triangle, the previous CPU methods need to use formula (6) for at least two edges. When data set is large, the efficiency will be greatly affected. Besides, for sorting methods, this process is independent and parallel for all triangles. But CPU has only few threads to run simultaneously. It is far more appropriate to use GPU to accomplish this work than CPU. GPU is very powerful for floating point computation and data parallel processing, which is good at dealing with numbers, matrixes and vectors. However, its efficiency may reduce when there are lots of logic operations, such as judgments and choices of branches. To optimize the algorithm for GPU, we design an edge table as follows:

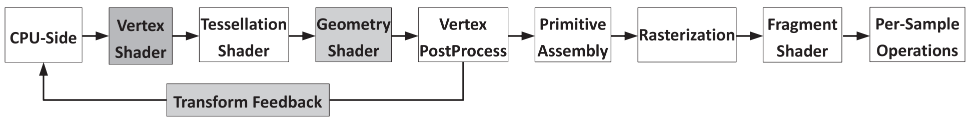


Fig. 7. Pipeline of modern GPUs.

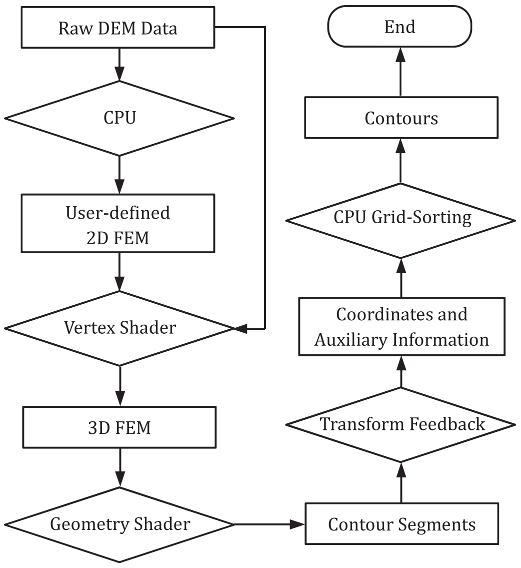


Fig. 8. Flow chart of our contouring method.

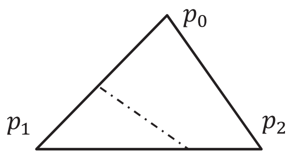


Fig. 9. Segment from a triangle.

Table 1
Edge table for segments extraction.

0	1	2	3	4	5	6	7
0	1	1	0	0	1	0	0
0	2	0	1	2	2	2	0
0	0	1	0	0	1	1	0
0	2	2	2	1	0	2	0

For OpenGL API 3.3 or higher, we can bind, interpret, and utilize any attribute for each vertex as needed. So in vertex shader, besides obtaining the 3D coordinates for FEM vertexes, we also bind an extra integer attribute b for each vertex, which will be 1 if the elevation value of the vertex is greater than the height of contour, otherwise 0. When it comes to geometry shader, for each triangle primitive, its three vertexes (p_0, p_1, p_2) can construct a three-digit binary number $(b_0b_1b_2)_2$. Obviously this number ranges from 0 to 7. So Table 1 contains every possible situation for segment extraction from a triangle, by which the existence of the possible segment and which edges are the segment generated from can be quickly determined.

Take Fig. 9 for example, if b_0 and b_2 are 0, b_1 is 1, then $(b_0b_1b_2)_2$ is 2. Look up in Table 1, and the corresponding array is (1,0, 1, 2), which means the edge $\overline{p_1p_0}$ and $\overline{p_1p_2}$ must contain contour points. So we can calculate their coordinates with formula (7). The other situations can be verified in the same way.

By using the edge table, we transform conditionals into array queries. So the efficiency on GPU can be increased. At the same time, by controlling the order of the array in table column, we can also ensure that the resulting segments have the same direction. For instance, the contour segments generated according to our edge table will be lower to the left and higher to the right. So it will be easier for further sorting.

The edge table can be stored as an 8×1 2D rectangular texture in GPU memory, with the format of 4 channels (RGBA), 8-bits unsigned integer. We can access it in the geometry shader through the texture coordinate $(b_0b_1b_2)_2$.

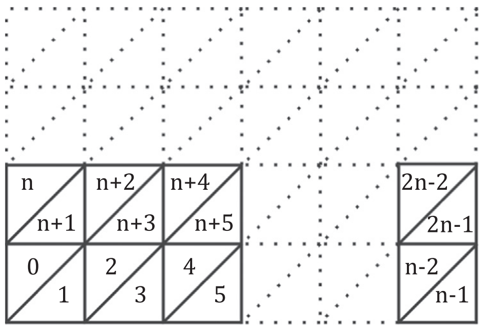


Fig. 10. The sequence of our FEM.

Also in geometry shader, we can define what to transmit back to the CPU-side with the transform feedback feature. For each primitive (triangle for this article), we can decide how many vertexes, and what attributes for these vertexes to transmit back. In our case, a segment from the triangle has 2 vertexes. Getting the 3D coordinates of them is the first thing we need.

Moreover, we also need to know the relationship between the segment and the FEM for efficient sorting. The FEM comes into the graphics pipeline with an internal sequence, primitive by primitive. The sequence we adopt is shown in Fig. 10, which is counted from left to right, bottom to top.

The geometry shader processes all of the primitives in parallel. But when processing each one of them, its sequence number of all the primitives can be queried by a built-in variable – *gl_PrimitiveIDIn*, for OpenGL 3.3 or highter. We send back this number too, as a 32-bits unsigned integer, if the primitive has a segment in it. As shown in Fig. 10, the position of the segment can be calculated directly according to this sequence number.

The GLSL codes for geometry shader are as follows:

```
//the texture of edge table
uniform sampler2DRect EdgeTex;
//the elevation of the contour
uniform float IsoValue;
//coordinates for 3 vertexes of the primitive(triangle), from the
  vertex shader
in vec3 pos[];
// extra attribute b for 3vertexes, b = 1 if pos.z > IsoValue,
  otherwise 0, from the vertex shader
in unsigned int b[];
//coordinate and sequence number to feedback, if there is a
  segment in this primitive
out vec3 feedback_position;
out unsigned int feedback_index;
vec3 CalcIntersection(vec3 Pos0,vec3 Pos1)
{
  float t = (IsoValue- Pos0.z)/( Pos1.z- Pos0.z);
  return mix(Pos0,Pos1,t);
}
void main()
{
  int l_nIndex = (b[0]«2)|(b[1]«1)|b[2];
  if(l_nIndex > 0 & & l_nIndex < 7)
  {
    vec2 l_TexCoord = vec2(l_nIndex+0.5, 0.5);
    ivec4 l_edge = ivec4(texture(EdgeTex, l_TexCoord));
    feedback_position = CalcIntersection(pos[l_edge.x], pos
      [l_edge.y]);
    feedback_index = gl_PrimitiveIDIn;
    //end feedback for one vertex
```

```
      EmitVertex();
      feedback_position = CalcIntersection(pos[l_edge.z], pos
        [l_edge.w]);
      feedback_index = gl_PrimitiveIDIn;
      //end feedback for another vertex
      EmitVertex();
      //end feedback for one primitive
      EndPrimitive();
    }
  }
```

After the geometry shader executes, its outputs can be received in CPU-side with certain OpenGL APIs – i.e. the coordinates and sequence numbers of the segments. The outputs also follow the input order of the FEM, but just for the primitives that contain segments. Render the FEM through this pipeline once, then we can get all the contour segments of a certain elevation. For contours of different elevations, the FEM needs to be rendered multiple times with different *IsoValues* in the shader.

3.3. Grid sorting

Just the segments are enough for rendering. But for more uses of GIS, we also propose a novel and efficient sorting algorithm based on the particular data structure generated by GPU. The basic idea of the algorithm is to associate the segments with their corresponding triangles in the FEM, then use the triangles for marking and sorting. The segments will be sorted and all of the contours will be composed by travelling the segments for just one pass.

The steps of our sorting algorithm are as follows:

① For each triangle, the necessary information is the potential segment in it, and a mark that records whether this triangle or segment has been travelled during the sorting process. So we construct a 2D array P according to the arrangement of triangles (Fig. 11), and each item P_i is a segment pointer. Set them to NULL as default. Also construct a 2D array of Boolean values, namely M. Set them to false as default.

② The result of transform feedback is a continuous region of memory, in which the coordinates and sequence numbers are stored separately, vertex by vertex (Section 3.2). Obviously two vertexes represent a segment. So by fetching the sequence number of each segment, we can assign the address of the segment coordinates to the corresponding pointer in P directly.

③ Then travel the segments. For each segment, find its corresponding triangle in step ①, namely i. If M_i is true, just skip it and travel to next segment. Otherwise, set M_i true, and go to step ④.

④ Now we can use this segment as the beginning of a contour. As shown in Fig. 11, the first found segment will be in triangle 2. Put the pointer P_i in an empty list, and record the first and last segments of the contour with the sequence numbers as S_f and S_l (both are 2 in the first place).

⑤ Assume the width of the FEM is n (n is 10 in Fig. 11). Then for triangle i, if i is odd, the indexes of its neighbors are $i-1$, $i+1$, $i-n-1$; otherwise, they are $i-1$, $i+1$, $i+n+1$.

So for the first and last segments of current contour, find their

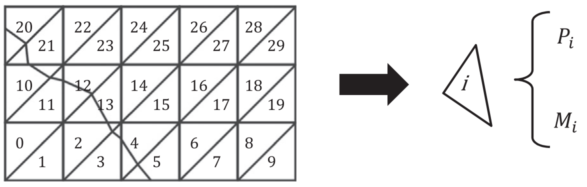


Fig. 11. Segments and triangles.

neighbor triangles. See if they have been travelled with M array. Skip the travelled triangles. For the rest of them, find out whether there are segments in them with P array.

For the neighbor triangles that don't have any, set their marks (M values) true.

When there are some segments found, check if any of them can be used to extend the contour. Since all of the segments have the same direction, according to Section 3.2, we only need to compare the coordinates of two points to find out whether the two segments are connected.

If there are any useful segments, push them in the front or back of the list according to the direction. Update S_f and S_l, and set M_{S_f}, M_{S_l} true as well. Otherwise, just skip them.

⑥ Repeat step ⑤, until this contour reaches the boundary of the FEM, or gets closure. The final resulting list is one complete contour line.

⑦ Go back to step ③, until all of the segments have been travelled. Then we can get all of the contours.

This process is much faster than the previous tracing method too. First, it doesn't need to travel all of the triangles. Just travelling the segments for one pass will be enough, which is a lot more efficient in most cases. So the complexity of our algorithm is $O(m)$, where m is the number of the segments. Second, in our only traversing pass, it doesn't need to solve whether there is a segment in the triangle, or to calculate the coordinates of the potential contour points anymore. Just some operations of comparison and assignment will be enough. Besides, the whole sorting process can be easily parallelized by different contour elevations, with just a line or two of OpenMP codes. So theoretically, this might be the most efficient routine.

4. Results

This method has been executed on a personal computer with an Intel E3 CPU (3.7 GHz, 8 threads) and a GTX 980Ti graphics card, on different DEM datasets (SRTM, ASTER, DLR, etc). The results are presented and analyzed as follows, from both aspects of appearance and efficiency.

The comparison of appearance between the previous CPU approaches and our method is shown in Fig. 12. When it comes to low-resolution DEM, the previous methods based on a FEM of the raw data can't achieve desirable results (Fig. 12.a). With the GPU C^3 continuous spatial interpolation and user-defined dense FEM presented in Section 2, we can illustrate every detail of the DEM data, and generate globally smooth contour lines (Fig. 12.b). Unlike the previous line-based smoothing algorithms, the method from this article is based on smooth interpolation of the terrain surface. So it will not cause any self-intersection at all.

For rendering only, there is no need for feedback to CPU and sorting. The segments can be extracted with only one rendering pass and then stored in VBO (Vertex Buffer Object, an OpenGL feature that allows vertex data to be stored in high-performance graphics memory without extra transfer to CPU-side) for visualization. This procedure can be very fast, within few milliseconds. Actually we can visualize dynamically changing contours in real time, according to the variation of the DEM data or the affecting parameters, such as the τ value in Section 2.1, the interval value of the contours, etc.

When it comes to tests of efficiency, the full sorting process should be performed. For the convenient comparison with the CPU approaches, here we construct a FEM with the same resolution of the raw DEM data, so the total amount of computations would be same. Some representative commercial software and open source projects are used for comparison, including ArcGIS, Global Mapper, GDAL and Surfer. Those four programs are most commonly available, and are widely used in industry and academia. Here ArcGIS and Global Mapper adopt the sorting strategy, the other two employ the tracing way.

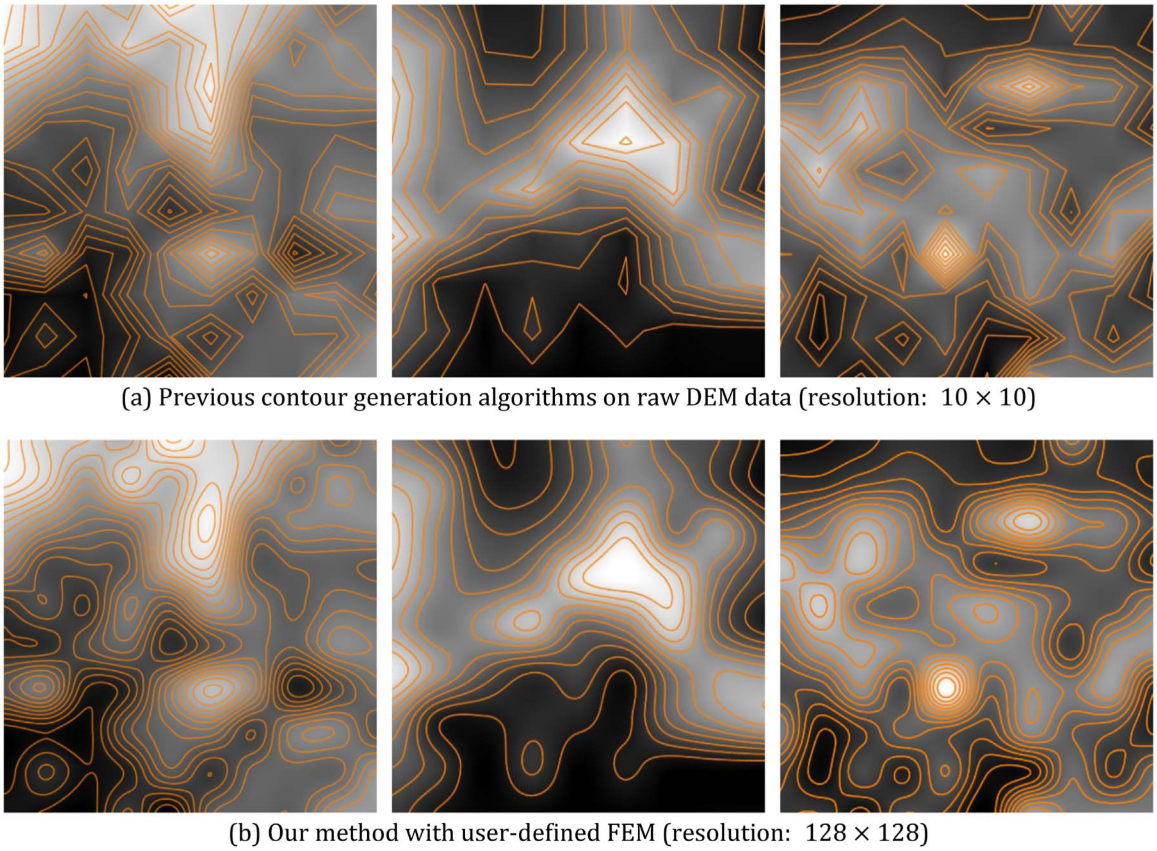

(a) Previous contour generation algorithms on raw DEM data (resolution: 10 × 10)
(b) Our method with user-defined FEM (resolution: 128 × 128)

Fig. 12. Comparison of smoothness.

Table 2
Comparison of efficiency.

Resolution of DEM Data		512×512	1024 × 1024	2048 × 2048	4096 × 4096
Time(*ms*)	**ArcGIS**	3117	5205	9522	40623
	Global Mapper	3131	6493	15031	33672
	GDAL	503	1273	3704	12948
	Surfer	452	1261	3482	12062
	Our Method	108	361	1293	4390

To gauge the performance more comprehensively, we run all of the applications with different DEM datasets, the resolutions of which vary from 512×512 to 4096 × 4096. For a given height interval -‐100 m, the results of time consumptions are shown in Table 2. We can see that our approach can be at least 2.6 times faster than the CPU based methods. For some implements, it may be tens of times faster.

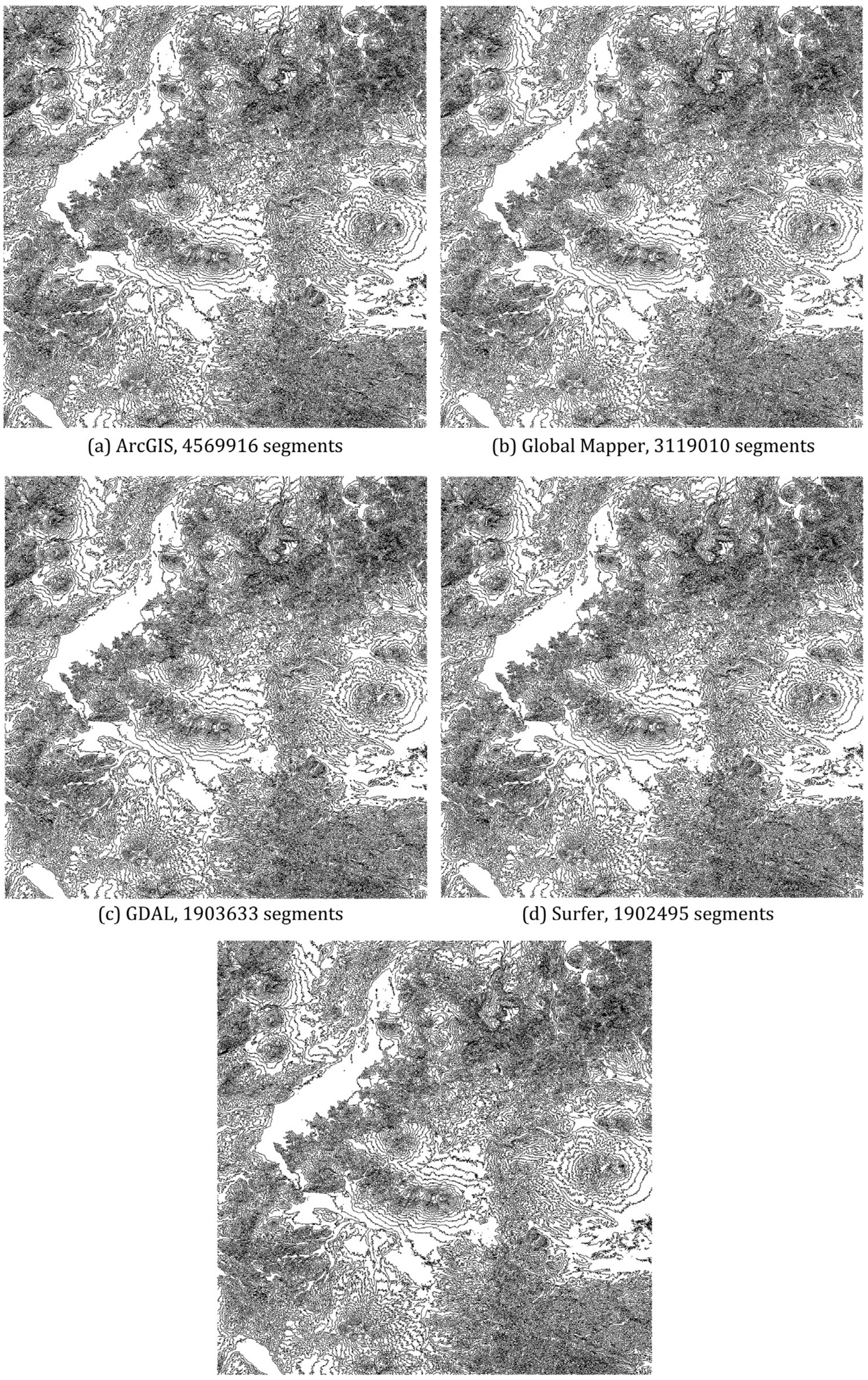
The complexity of contour generation algorithm can be uncertain for different datasets. Assume the number of sampling points in the DEM is n, and the number of potential segments is m. For a typical tracing algorithm, its complexity can't be less than $O(n)$; and for a sorting algorithm, its complexity can't be less than $O(n+m)$. Our approach consists of three major phases: segments extraction by geometry shader, data feedback to CPU-side, and the Grid Sorting. The first part belongs to a single frame, which only takes few milliseconds to perform. Then in the second part, time cost for segments transform feedback is considerable, which is proportion to the amount of extracted segments and is mainly concerned with the bandwidth of the graphic card. For the third part, Grid Sorting is an $O(m)$ algorithm (Section 3.3) and only needs to travel all of the segments once, whose time consumption is also proportion to the amount of segments. Table 3 shows the number of segments extracted from different datasets, and the detailed time consumptions for both GPU and CPU parts.

Considered as a whole, the time saved by geometry shader based segments extraction is far more than the cost caused by transform feedback, and the resulting segments can greatly reduce the time complexity of the sorting algorithm in CPU-side. Thus we can achieve high efficiency, as the results confirm.

Fig. 13 shows the resulting contours from different GIS programs and our method, based on the same DEM data with 4096 × 4096 resolution. The shapes of different results look quite similar in general. However, GDAL and Surfer find much less segments than the others, which means some details must have been lost. This may also help to explain why they are much faster than ArcGIS and Global Mapper to a

Table 3
Details of time consumption for our approach.

Dataset Index	1	2	3	4	5
Resolution	2048 ×2048	2048×2048	4096×4096	4096×4096	4096×4096
Number of Segments	122641	133052	690670	638683	516003
GPU Time (*ms*)	38	38	176	158	107
CPU Time (*ms*)	69	77	728	707	457



(a) ArcGIS, 4569916 segments

(b) Global Mapper, 3119010 segments

(c) GDAL, 1903633 segments

(d) Surfer, 1902495 segments

(e) Our Method, 3211641 segments

Fig. 13. Contouring results of **4096×4096** DEM data from different methods.

certain degree. We can see that our method can generate contours that retain full details of the raw DEM data. In the meantime, it achieves the highest time efficiency.

The binary executable examples of our algorithm and the datasets for the tests above can be downloaded from https://github.com/starforce08/GPUContouring. The other programs are quite widely used, which can be easily found in internet. However, to fulfill the tests, several commercial licenses may need to be purchased.

5. Conclusions

We have surveyed the previous contouring methods, and introduced the GPGPU technology to solve the problems of appearance and efficiency innovatively. The method presented in this paper makes use of multiple stages of modern GPU pipeline, which interpolates the raw Grid DEM data with C^3 continuity by vertex shader, and can generate globally smooth contour lines. The contour segments are extracted by geometry shader, then transferred as a feedback to CPU-side, and sorted into lines without any extra pre-processing calculation. Besides, the resulting contours are vectors, which can be easily used in various applications. The algorithm has very high efficiency, and is easy to implement on most of the current graphic hardware, which can support the OpenGL 3.3 API or higher. Moreover, GPU pipeline can do more than parallel computation. The tessellation shader and compute shader (Fig. 7) also have unique functions, which may play important roles in contour refinement. We intend to research on this field in the following study.

Our work might also raise and inspire some future thinking about other topics around geographic analysis and geo-visualization, with the new possibilities provided by the enormous, highly parallel computation power of the modern GPU architecture.

References

Akima, H., 1978. A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. ACM Trans. Math. Softw. 4 (20), 148–159.

Barcha, J.P., Reese, J.R., 1964. Surface determination and automatic contouring for mineral exploration, extraction and processing. Colo. Sch. Mines Q. 14 (4), 187–194.

Buys, J., Messerschmidt, H.J., Botha, J.F., 1991. Including known discontinuities directly into a triangular irregular mesh for automatic contouring purposes. Comput. Geosci. 17 (7), 875–881.

Bjorke, K., 2004. High-Quality Filtering. In GPU Gems, Randima Fernando Ed. Addison-Wesley, pp. 391–415.

Bryan, S.M., Liu, W.M., Terry, S.Y., Kalpathi, S., 2005. Active contours using a constraint- based implicit representation. In: Courses of ACM SIGGRAPH 2005, Los Angeles, California, 252-259.

Chester, R.P., Thomas, A.E., Boyd, H.A., 1968. Automatic contouring of irregularly spaced data. Geophysics 33 (3), 424–430.

Chui, C.K., 1988. Multivariate Splines. Capital City Press, Montpelier, Vermont, 198.

Cheng, J.M., Chen, C.X., Sun, H.L., 1998. Automatic generation of contour in triangular mesh and its programming. J. Hydraul. Eng. 16, 23–26.

Carsten M., 2003. Geometric designing and space planning using the marching squares and marching cube algorithms. In: International Conference on Geometric Modeling and Graphics, Proceedings, 90-95.

Christian, S., Markus, H., 2005. Fast third-order texture filtering. In: Pharr, M. (Ed.), In GPU Gems 2. Addison-Wesley, Boston, Massachusetts,USA, 313–329.

Chen, Z., Shen, L., Zhao, Y.Q., Yang, C.J., 2010. Parallel algorithm for real-time contouring from grid DEM on modern GPUs. Sci. China Technol. Sci. 53 (1), 33–37.

Franke, R., 1982. Scattered data interpolation: tests of some method. Math. Comput. 38 (157), 181–200.

Grain, I.K., 1970. Computer interpolation and contouring of two-dimensional data: a review. Geoexploration 8 (2), 71–86.

Goldin, D., Gao, H.Y., 2006. Dynamic isoline extraction for visualization of streaming data. Lect. Notes Comput. Sci. 3967, 415–426.

Gul, S., Khan, M.F., 2010. Automatic Extraction of Contour Lines from Topographic Maps. In: International Conference on Digital Image Computing: Techniques and Applications (DICTA), Sydney, NSW, pp. 593–598.

Hou, F.Z., Zhuang, J.J., Ning, X.B., 2008. Iso line drawing algorithm and programming implementation based on case table. J. Nanjing Univ. (Nat. Sci.). 44 (4), 371–378.

Hiremath, P.S., Kodge, B.G., 2010. Generating contour lines using different elevation data file formats. Int. J. Comput. Sci. Appl. (IJCSA). 3 (1), 19–25.

Jaara, K., Lecordix, F., 2011. Extraction of cartographic contour lines using digital terrain model (DTM). Cartogr. J. 48 (2), 131–137.

Kok, R., Begin, J., 1981. Evaluation of automatic contouring methods for drainage design. Trans. ASAE 24 (1), 87–96, (102).

Kolingerová, I., Dolák, M., Strych, V., 2009. Eliminating contour line artefacts by using constrained edges. Comput. Geosci. 35 (10), 1975–1987.

Lodwick, G.D., Whittle, J., 1970. A technique for automatic contouring field survey data. Aust. Comput. J. 2 (3), 104–109.

Lee, S.Y., Wolberg, G., Sung, Y.S., 1997. Scattered data interpolation with multilevel B-splines. IEEE Trans. Vis. Comput. Graph. 3 (3), 228–244.

Li, Z.L., Zhu, Q., 2003. Digital Elevation Model 2nd edn. Wuhan University Press, Wuhan, 203.

Lalescu, C.C., 2009. Two hierarchies of spline interpolations: practical algorithms for multivariate higher order spline. Comput. Sci. Prepr. arXiv 0905, 3564.

Li, W., Li, S.Y., 2010. Generation methodology of isolines of earthquake ground motion. Earthq. Eng. Eng. Vib. 9, 473–480.

Miao, R.Z., 2004. A new smoothed contour lines generating algorithm for quadrilateral meshes. J. Chang. Univ. Sci. Technol. 27 (1), 17–18.

Nielson, G.M., 1993. Scattered data modeling. IEEE Comput. Graph. Appl. 13, 60–70.

Nickerson, B.G., Judd, P.A., Mayer, L.A., 1999. Data structures for fast searching of SEG-Y seismic data. Comput. Geosci. 25 (2), 179–190.

Norman, L.J., Michael, J.K., Alan, K.Z., 2000. Fast algorithm for generating sorted contour strings. Comput. Geosci. 26 (7), 831–837.

Paul, K., 1988. A thinning algorithm by contour generation. Commun. Acm. 31 (11), 1314–1324.

Peters, R., Ledoux, H., Meijers, M., 2014. A Voronoi-based approach to generating depth-contours for hydrographic charts. Mar. Geodesy. 37 (2), 145–166.

Qian, C., Du, Z.H., Cao, R.Z., Zhang, F., Liu, R.Y., 2014. Research of parallel global sea surface temperature contours extraction algorithm on CUDA platform. J. Zhejiang Univ. (Sci. Ed.). 41 (1), 82–89.

Rognant, L., Planès, J.G., Memier, M., Chassery, J.M., 2001. Contour lines and DEM: generation and extraction. Lect. Notes Comput. Sci. 2181, 87–97.

Riegler, G., Hoeppner, E., Li, X., 2006. Automatic contour line generation using intermap's digital terrain model. In: Proceedings of ASPRS Annual Conference, Reno, NV, May, pp. 1–5.

Rui, X.P., Song, X.F., Yang, Y.G., Ju, Y.W., 2011. An improved method of rendering oil extraction contour under constrained conditions. Min. Sci. Technol. 21 (3), 337–342.

Schoenberg, I.J., 1969. Cardinal interpolation and spline functions. J. Approx. Theory 2 (2), 167–206.

Schoenberg, I.J., Sharma, A., 1973. Cardinal interpolation and spline fucntions V. The B-splines for cardinal Hermite interpolation. Linear Algebra Appl. 7 (1), 1–42.

Schmieder, A., Huber, R., 2000. Automatic Generation of Contour Lines for Topographic Maps by Means of Airborne High-Resolution Interferometric Radar Data. In: Proceedings of ASPRS Annual Conference, Washington, DC, pp. 5420–5422.

Sun, G.R., Ma, L., Lu, D.P., Zhao, G.R., Hao, J.L., 2000. Investigation on the algorithm of making and filling isoline. J. Tianjin Univ. 33 (6), 816–818.

Schlei, B.R., 2009. A new computational framework for 2D shape-enclosing contours. Image Vis. Comput. 27 (6), 637–647.

Shao, L., Dong, G.L., Liu, J., Mu, Y., Guo, P., 2014. Grid Sequence Algorithm Generating Contour Based on Delaunay Triangulation. In: Proceedings of the 11th IEEE International Conference on Mechatronics and Automation, Tianjin, China, pp. 2011–2016.

Van, K., , 1996. Efficient methods for isoline extraction from a TIN. Int. J. Geogr. Inf. Syst. 10 (5), 523–540.

Watson, D.F., 1982. ACORD: automatic contouring of raw data. Comput. Geosci. 8 (1), 97–101.

Watson, D.F., 1993. Contouring: A Guide to the Analysis and Display of Spatial Data. Elsevier, Pergamon, Oxford, 321.

Wang, Z., Liu, H.W., Chai, H.J., 2001. Automatic generation of contours in raster graphics. J. Hydraul. Eng. 4, 53–56.

Wang, X.L., 2006. The research of automatic establishing contour and contour smoothing. China University of Petroleum, Qingdao, Shandong, China, 105.

Wang, T., Liu, J.P., Wu, H.H., 2006. The extraction of contour lines from grid DEM based on sorting. Acta Geod. Cartogr. Sin. 35 (4), 52–56.

Wang, T., Wu, H.H., 2006. An algorithm for generating contour from gridded DEM. Sci. Surv. Mapp. 31 (2), 108–110.

Wang, J.C., Qian, C.H., Rui, Y.K., 2007a. A practical method for contour generation based on raster data. Sci. Surv. Mapp. 32 (6), 88–90.

Wang, T., Wu, H.H., Liu, J.P., 2007b. An algorithm for extracting contour lines based on interval tree from grid DEM. Geomat. Inf. Sci. Wuhan. Univ. 32 (2), 131–134.

Xie, S.P., Tian, D.S., 1995. To improve tracing isopleth by route grid method. Acta Geod. Cartogr. Sin. 24 (1), 52–56.

Xie, J.B., 2012. Implementation and performance optimization of a parallel contour line generation algorithm. Comput. Geosci. 49, 21–28.

Yates, S.R., 1987. CONTUR: a FORTRAN algorithm for two-dimensional high-quality contouring. Comput. Geosci. 13 (1), 61–76.

You, G.Z., 1989. The method for plotting contours on rectangular grid. J. East China Inst. Technol. 49 (1), 45–51.

Zhang, C.M., 1991. Automatic generation of contours in raster graphics. Chin. J. Comput. 03, 229–232.

Zhang, M.H., Liang, W.K., 1997. An improved method for contouring on network of triangles. J. CAD CG 9 (3), 213–217.

Zhang, Q., Li, P., Miao, J., 2001. A contour line generation method and its parallel realization. J. Huazhong Univ. 29 (9), 4–6.

Zhao, J.S., 2003. An adaptive algorithm of contour interpolation based on grid DEM. J. Cent. South Univ. (Sci. Technol.). 34 (3), 315–319.

Zhao, J.C., Bai, R.C., Liu, G.W., Liu, W., 2014. Fast isolines generation algorithm based on TIN. Comput. Eng. Appl. 50 (24), 10–15.