Interective Point Clouds Fairing on Many-Core System

Tang Jie
Department of Computer Science and Technology
Nanjing University
Nanjing, China
tangjie@nju.edu.cn

Wu Gangshan
Department of Computer Science and Technology
Nanjing University
Nanjing, China
gswu@nju.edu.cn

Xu Bo
Department of Computer Science and Technology
Nanjing University
Nanjing, China
xubo@graphics.nju.edu.cn

Gong Zhongliang
Department of Computer Science and Technology
Nanjing University
Nanjing, China
gzl@graphics.nju.edu.cn

***Abstract*—This Paper proposes an interactive point clouds fairing algorithm running on many-core system. The algorithm is composed of four steps. Firstly, a *k* nearest neighbor searching method was designed which could fully utilize the computing ability of GPU. Secondly, a parallel Gaussian weighted normal estimation was put forward. Thirdly, a weighted fairing method was proposed to get better result especially for the unevenly distributed point clouds. The whole algorithm was implemented on NVIDIA GPU using CUDA. Experimental results show that the algorithm could achieve interactive fairing of large size point clouds with good quality.**

Keywords- point clouds; fairing; CUDA; GPU

I. INTRODUCTION

Point clouds have become increasingly popular in modeling. Due to improved graphics hardware and technologies for the acquisition of point geometry, point clouds are getting larger and larger. Point clouds has simple data description and easy to get. Therefore, it is widely used in many fields. However, there usually exist some noise in point clouds due to the fluctuation during the measure course. Hence, it is critical to erase the noise as much as possible. Otherwise, it will influence the quality of following processing.

Fairing is used to eliminate or alleviate the noise 3d model. So fairing effect is an important feature of fairing algorithm. Unlike meshes, there is no explicit connection information in point clouds. Furthermore, it is very easy to mistake the connection status nearby sharp features such as thin parts, which will further influence the effect of fairing. Running speed is an important feature of fairing algorithm. Real time or interactive fairing is the most favorable. Unfortunately, with the growth of data size of point clouds, traditional fairing methods such as Laplace filtering etc, which perform fairing iteratively, usually cost several minutes or more on a model with 100,000 points. This is unacceptable in interactive modeling system. So we have to find a better solution.

Modern graphics processing units(GPUs) have been at the leading edge of increasing chip-level parallelism for some time. Current NVIDIA GPUs are many-core processor chips, scaling from 8 to 240 cores. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth. This degree of hardware parallelism reflects the fact that GPU architectures evolved to fit the needs of those problems with tremendous inherent parallelism, such as point clouds fairing. However, this also raised many important questions about how to productively develop efficient parallel programs. Exist point fairing methods cannot be shifted to many-core system directly.

In this paper, we extent the bilateral mesh denoising method proposed by Fleishman[1] to point clouds. The fairing process is divided into 4 steps: uniform grid construction, k-neighbor search, normal estimation and fairing. Each step is converted into a kernel function which could be executed paralleledly on GPU. Our contributions are as follows:

- Extract the parallelism in bilateral point denoising algorithm, execute it on a many-core system and achieve interactive point fairing.
- Propose a new parallel k nearest neighbour searching algorithm with good quality and fast speed.
- Improve the normal estimation method to deal with unevenly distributed points.
- Proposed a weighted bilateral denoising method to cope with unevenly distribute points.

Our fairing algorithm has been implemented on a graphics card using CUDA[2] for a significant speedup over the CPU implementation.

II. RELATED WORK

According to the diffusion type, point clouds fairing could be categorized into two types: isotropic method and anisotropic method. According to fairing evenly or not, point clouds fairing could be categorized into linear fairing and non-linear fairing. According to the filter type, point cloud fairing could be categorized into the following groups: Laplace fairing, Wiener filtering, bilateral denoising, and Moving Least squares method.

Laplace fairing diffuses high frequency noise to achieve filtering[3][4]. In fact, Laplace filtering is the minimization of energy of a surface. With the increase of iterations, the shrinking and distortion are inevitable. To overcome this problem, Vollmer[5] adjust the vertex's position along its normal with a little offset after it has been faired. But distortion nearby sharp features still exists. Those methods belong to isometric linear fairing, in which denoising and feature keeping are contradictive.

978-0-7695-4190-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ISPA.2010.49

Peng[6] apply the local adaptive filtering to the mesh and point denoising, but the local connectivity should be aware of before fairing. Alexa[7] proposed a Wiener filter utilized in mesh denoising. Similar to Laplace filter, the method also diffuses the high frequency noise into local neighbor and has to know the local connectivity.

Moving Least squares(MLS)[8] belongs to surface fitting method based on iteration. For the input point clouds, firstly, an approximating hyperplane has to be determined for points p near the (d -1)-dimensional hypersurface S which is sampled by {pi}. This is done by solving a non-linear minimization problem. Then the hypersurface is interpolated locally by polynomials that have the hyperplane defined in the first step as domain. The non-linear minimization problem is reformulated as an eigenvalue problem of an associated weighted covariance matrix. The second step is a system of linear equations whose size depends on the degree of the approximating polynomials. The main idea of this method is to implicitly define an approximating surface. MLS perform well on denoising, but acted bad where the shape is sharp.

Different from Laplace filtering, bilateral filtering use weighted summation of neighbor sample points to adjust the fairing point, which could preserve the feature very well. Fleishmann[1], Jones[9] utilize it to achieve mesh denoising. The method does not need iterations, and could keep the sharp feature pretty well.

Actually, the fairing of large point clouds runs very slow. And the fairing could be done parallelly. Jalba[10] proposed a mesh smoothing method using GPU, which runs pretty fast. Ni[11] gave a quadrangle mesh fairing method. These methods are both cope with meshes which have explicit connectivity.

Bilateral denoising does not need iteration and could keep sharp feature while eliminate the noise. However, existed method could only process meshes which have explicit neighbor connectivity. Also while processing unevenly mesh fairing, it is straightforwardly to use the area of triangle as weights. But pointset does not have such attribute to use. Bilateral denoising method does fairing point by point. Hence the method possesses high parallelism. However, the sub-process, such as k-neighbour searching is time consuming with less apparent parallelism. This section will introduce our interactive point clouds fairing method suitable for many-core system. The experimental results will be discussed in next section.

III. EASE OF USE

A. Bilateral mesh denoising

Bilateral mesh denoising[1] is a diffusion method, which move each vertex of a mesh along its normal for an offset. The offset is calculated from the vertex's neighbor geometry, as follows:

$$v_i^{'} = v_i - n_i \cdot d_i \qquad (1)$$

$$d_i = \frac{\sum_{v_j \in N(v_i)} n_i \cdot (v_i - v_j) \cdot W_c(\|v_i - v_j\|) \cdot W_s(n_i(v_i - v_j))}{\sum_{v_j \in N(v_i)} W_c(\|v_i - v_j\|) \cdot W_s(n_i(v_i - v_j))} \qquad (2)$$

Where n_i is the normal of v_i, $N(v_i)$ is the neighborhood of v_i. In practice, $N(v_i)$ is defined by a set of points $\{q_i\}$, where $\|v_i - q_i\| < r = 2\sigma_c$. The closeness smoothing filter is the standard Gaussian filter with parameter σ_c: $W_c(x) = e^{-x^2/(2\sigma_c^2)}$, and a feature-preserving weight function, which refered to as a *similarity weight function*, with parameter σ_s that penalizes large variation in intensity, is: $W_s(x) = e^{-x^2/(2\sigma_s^2)}$.

Unfortunately, there is no explicit connectivity information in point clouds. Therefore, we have to compute it from the local area of each point. To best utilize the computing ability of many-core system, the whole fairing process is divided into 4 sub processes with great parallelism as follows:

- Constructing fast space indexing data structure for point clouds
- K-neighbour searching
- Normal estimation
- Bilateral denoising

B. Construction of Uniform Grid

Uniform grid is used for fast indexing of space point. First the bounding box of all points is divided into cells with equal size, and the index of each cell could be calculated by its position. During the initializing period, each point is allocated into a cell according to its position. After that, we could fast retrieve all points from a given cell.

We have implemented our algorithm on NVIDIA GPU using CUDA technology. CUDA does not support dynamically allocate memories in kernel function. However the number of points in each cell is variable. Hence we have to design a flexible data structure to solve the problem. Fig. 1 is the data structure of point clouds on device end.

```
float3 *  d_vertex;       //original point clouds
int *     d_grid_head;    //the index of the first point in each cell
                          of //uniform grid
int*      d_vertex_link;  //the list of the other points in each cell
```

Figure 1. Data structure of point clouds on CUDA

Fig. 2 shows an example of uniform grid with 4 cells. 8 points are distributed into the grid (Fig. 2b). d_vertex stores the position information of 8 points. d_grid_head stores the indices of head points in each cell. d_vertex_link stores the indics of other points in each cell until the index is -1.

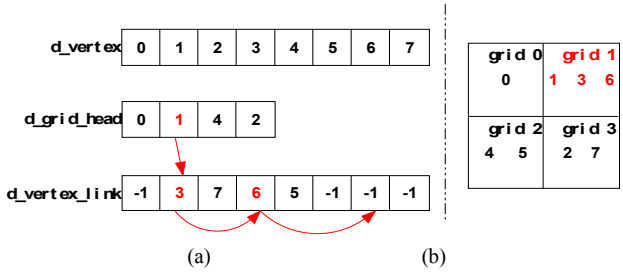

Figure 2. Sample of Data structure of point clouds on CUDA

Algorithm 1 shows the pseudo code of kernel function which is executed in device end. Because some threads may write the same data in d_grid_head simultaneously, antomic operation atomicExch() is used to exchange two items.

```
Algorithm 1: uniform grid construction algorithm
__global__ void UniformGrid(){
    int vidx = blockIdx.x*blockDim.x+threadIdx.x;
    if(vidx>vnum) return;
    conmpute gidx from the point position and
bounding vox;
    d_vertex_link[vidx]=atomicExch(&d_grid_head[gi
dx],vidx);
}
```

C. *k* Nearest Neighbour Searching

Computation of k Nearest Neighbour forms a basic building block in solving many important problems including normal estimation, surface simplification, finite element modeling, shape modeling and surface reconstruction. With the growing sizes of point clouds, the emergence of many-core processors in mainstream computing and the increasing disparity between processor and memory speed, it is natural to ask if one can gain from the use of parallelism for the k-Nearest Neighbour construction problem.

K Nearest Neighbour searching has great parallelism, very suitable for GPU computing. Garcia[12] and Liang[13] proposed algorithms using GPU to accelerate the computing of KNN. However, both of them belong to naive approach and run slower when dealing with large datasets with high dimension. We have implemented a KNN searching method based on uniform grid, which running fast on GPU. Algorithm 2 shows the pseudo code of its kernel function.

While searching k nearest neighbor for point v, the method acts as follows: Starting from the cell where the point v lies, the distances between v and all other points lie in the cell are computed and sorted. If the number of the nearest neighbor is less than k, the searching range will increase in three directions by one cell. Compute the distances from v to six bounding planes. If one of the distances is less than the distance from v to found neighbor, increase the searching range in this direction by one cell. Figure 3 illustrate the circumstance. Cell B is the current searching range, s is one of the nearest neighbor of v. However, the distance from v to the left plane of B is less than that from v to s. So there exists the possibility that some point in cell A is nearer to v than s is. Hence the method increases the searching range in that direction by one cell.

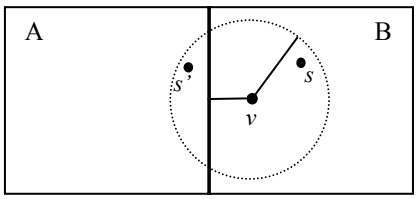

Figure 3. *k* Nearest Neighbour searching

Using Euclidean distance to sort the neighbor points sometime will lead to incorrect results near sharp features. Fig. 4 gives an example. There are two lines of sample points. When we compute k nearest neighbor points of v_1, we will find that v_2 is nearer than v_3. But using v_2 to compute the normal of v_1 will lead to an incorrect result. So we have to find a way to improve it. If the point clouds is assumed to be a ρ-dense, δ-noise, when we add a new point q into the found nearest neighbor point set, we make a rule that $d(q, N) < \rho+\delta$, where N is neighbor point set of v which has been searched, and d is minimum distance between q and N. So each neighbor point of v will not cross the sharp fearture, and the normal will be more precise.

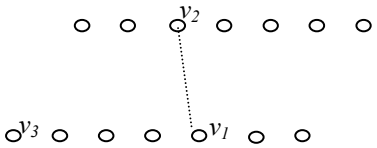

Figure 4. Constrained *k* Nearest Neighbour searching

D. *Normal Estimation*

Bilateral denoising method moves the point along its normal direction for a small offset, which will only fair the geometry of the point cloud and keep the parameterization unchanged. The correctness of normal direction is critical to bilateral denoising. Hoppe[14] computed the least squares plane of a point and use the normal of that plane as the normal of the point. Alexa[8] adopted the moving least squares method, which is more precise but need too much computation.

Hoppe's method runs fast and is appropriate for GPU computing. During the implementation, we noticed that Hoppe's method will get unexpected results while coping with unevenly distributed point clouds. Fig 8 shows an example. Fig 8a is the original point set. In Fig 8b, if there is a noise point such as the one in the white rectangle, the computed normal will point to an unexpected direction. To solve the problem, we propose a new Gaussian weighted least squares method to estimate normal from noisy point cloud. The method first constructs the covariance matrix like equation (3). Then the eigenvector corresponding to the

least eigenvalue of the covariance matrix is selected as the normal of the point.

$$M = \sum_{v_j \in N(v_i)} (w_g(\| v_j - v_c \|)(v_j - v_c)^T)(w_g(\| v_j - v_c \|)(v_j - v_c)) \quad (3)$$

$$v_c = \sum_{v_j \in N(v_i)} w_g(\| v_j - v_i \|) v_j \quad (4)$$

$$w_g(x) = e^{-x^2/(2\sigma^2)} \quad (5)$$

E. *Fairing*

Section III.A introduced the bilateral mesh denoising method. However, we find that BMD method performs not so good when it processes unevenly dense point clouds. This is because it adopted an unweighted method to calculate the offset of each point. Dense part will contribute more to the computed offset than the sparse part will. Hence the parameterization will affect the result greatly. Therefore, it is better to design a weighted method to compute the offset of a point only from its local geometric shape. Unfortunately, point clouds do not have an inherent property to be used as a weight, while the local area could be used as a weight in mesh denoising.

In this paper, we proposed a weighted fairing method of point clouds. For each point $\boldsymbol{v}$ in the point clouds, an average length for $\boldsymbol{v}$ to its 5 nearest neighbor is computed. We use the length's square as the weight. The equation (2) is improved as:

$$d_i = \frac{\sum_{v_j \in N(v_i)} n_i \cdot (v_i - v_j) \cdot W_c(\|v_i - v_j\|) \cdot W_s(n_i(v_i - v_j)) W_a(v_j)}{\sum_{v_j \in N(v_i)} W_c(\|v_i - v_j\|) \cdot W_s(n_i(v_i - v_j)) W_a(v_j)} \quad (6)$$

Where $W_a(\boldsymbol{v}_j)$ is the average length from $\boldsymbol{v}_j$ to its five nearest neighbors. Computing average length for each point on CPU is time consuming especially for large scale data. However, when we turn to GPU, we could get an interactive result for a point clouds containing a million points.

IV. EXPERIMENTAL RESULTS

We have implemented the algorithm using CUDA 2.3 on MS Visual Studio 2008. Two GPU devices were tested. One is NVIDIA GeForce 9800GT, the other is NVIDIA Tesla 1060. The specifications of the two devices are listed in the following table.

TABLE I. TEST ENVIRONMENT

	GeForce 9800GT	***Tesla 1060***
core	112	240
Processor Clock	1500MHz	1.296GHz
Memory	512MB	4.0GB
Memory Bandwidth	57.6 GB/s	102GB/s

We have test many models. Following are two of them. The left column shows original model, and the right one shows the noisy one.

Figure 5. Buddha and its noisy model

A. *k nearest neighbor searching*

The running time of k nearest neighbor searching is affected by the resolution of uniform grid and the parameter k. Figure 6 shows the uniform grid construction time of Buddha which has 543652 points. From the figure, we could find that the construction time is related to the number of computing core of GPU. Figure 7 shows the time cost by k nearest neighbor searching.

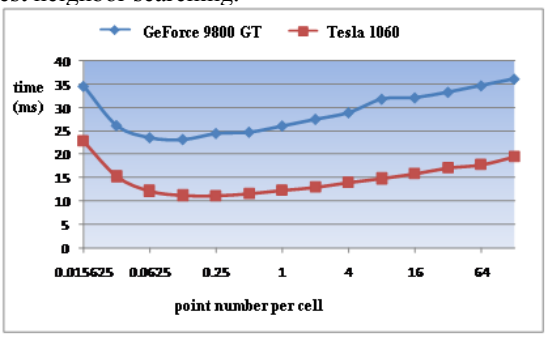

Figure 6. Uniform grid construction

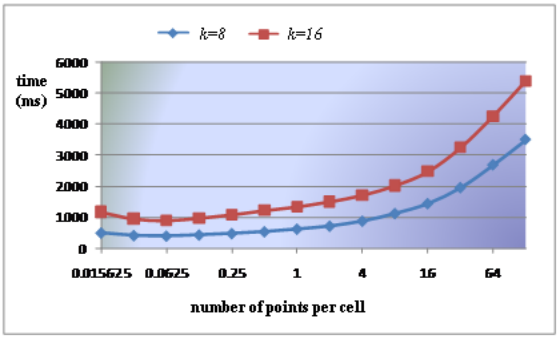

Figure 7. k nearest neighbor searching

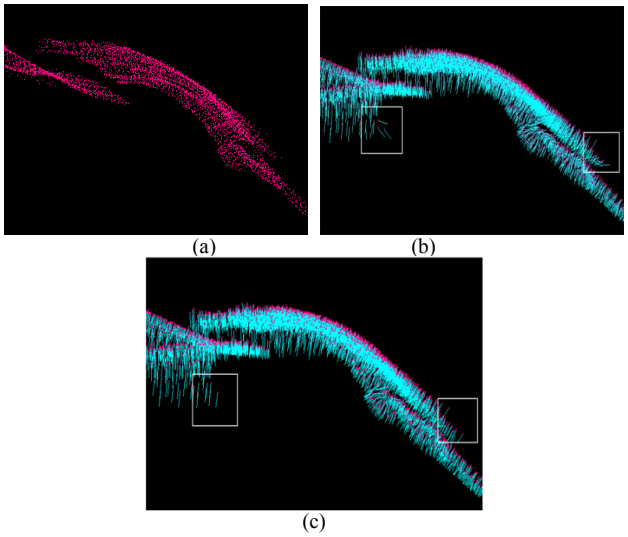

Figure 8. Constrained *k* nearest neighbor searching

Figure 8 shows the result of constrained k nearest neighbor searching. Figure 8a is the original point set. Figure 8b shows the computed normal using old method. We could see than the normals in the white box are not properly computed. The reason is explained in the section 3.3. Figure 8c shows the improved results of constrained nearest neighbor searching. The normals in white box were computed more precisely.

B. Fairing

Table 2 shows the running time of our fairing method. K is set to be 8. And the time includes transferring initial data from RAM to GPU through PCI, fairing and transfer back the result. Most existed fairing algorithm focus on the fairing quality, so we find little introduction on time consuming of those method. But we have implemented some of those methods. It seems that those methods run pretty slow when dealing with large scale dataset. But from the table 2, we could find that for the model has almost half million points, we could achieve interactive fairing even using a common graphics card like GeForce 9800GT. Figure 9 shows the fairing results of Armadillo. Left is the noisy model and the right is the fairing result.

TABLE II. FAIRING TIME

Num of point	12683	50833	114446	458074	1272739
CPU(s)	23	106	250	719	1439
GrForce 9800GT(s)	0.135	0.533	1.202	4.954	14.682
Tesla 1060(s)	0.027	0.099	0.215	0.883	2.617

Figure 9. Armadillo and its noisy model

V. CONCLUSION AND FUTURE WORK

This paper proposed a parallel point clouds fairing method using NVIDIA CUDA. The method takes the full advantage of the high computing ability of GPU. The whole fairing course is divided into four separate processes with great parallelism, which are uniform grid construction, *k* nearest neighbor searching, normal estimation and fairing respectively. In order to overcome the inaccuracy caused by unevenly distributed point clouds, the k nearest neighbor searching method, normal estimation and fairing method are improved. Interactive results could be achieved on common graphics card for large scale dataset.

Point clouds are getting larger and larger. Therefore, it is better to do the processing while transferring the data. Streaming fairing is what we want to do in the near future. How to fully utilize the share memory is another research subject attracting our attention. Using share memory could accelerate the algorithm greatly. But parallelism is difficult to manage.

ACKNOWLEDGMENT

Work on this paper was partially supported by the Natural Science Foundation of Jiangsu Province (No. BK2008262) and the National High-Tech Research and Development Plan of China under Grant No. 2007AA06A402

REFERENCES

[1] Fleishman S, Drori l, Cohen-Or D. Bilateral mesh denoising. Proceedings of ACM SIGGRAPH 2003, 950-953.

[2] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. http://developer.nvidia.com/cuda, Jan. 2007.

[3] G Taubin. A signal processing approach to fair surface design. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, Los Angeles, California, 1995, 351-358.

[4] M Desbrun, M Meyer, P Schröder, et al. Implicit fairing of irregular meshes using diffusion and curvature flow. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, Los Angeles, California, 1999, 317-324.

[5] Vollmer J, Mencl R, Muller H. Improved Laplacian smoothing of noisy surface meshes. EUROGRAPHICS 1999 Conference Proceedings, 1999,131-138.

[6] Peng J, Strela V, Zorin D. A simple algorithm for surface denoising. Proceedings of the conference on Visualization 2001, San Diego, California, October, 2001, 107-112.

[7] Alexa M. Wiener filtering of meshes. Proceedings of Shape Modeling International, Calgary, Aiberta, 2002, 51-57.

[8] Alexa M, Behr J, Cohen-Or D et al. Computing and rendering point set surfaces. IEEE Transactionson On Visualization and Computer Graphics, 2003, 9(1):3-15.

[9] Jones T R, Durand F, Desbrun M. Non-iterative, feature-preserving mesh smoothing[C]. SIGGRAPH 2003, 943-949.

[10] Jalba, A.C., Jos B.T.M.Roerdink. Efficient surface reconstruction from noisy data using regularized membrane potentials. IEEE Transactions on Image Processing, 2009,18(5):1119-1134.

[11] Ni,T., Yeo,Y., Myles,A., Goel,V., Peters,J.. GPU smoothing of quad meshes. IEEE International Conference on Shape Modeling and Applications, Stony Brook, NY, June, 2008, 3-9.

[12] Garcia,V., Debreuve,E., Barlaud,M.. Fast k nearest neighbor search using GPU. Computer Vision and Pattern Recognition Workshops, Anchorage, AK, June, 2008, 1-6.

[13] Shenshen Liang, Cheng Wang, Ying Liu, Liheng Jian. CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU. Information, Computing and Telecommunication , Beijing, 2009, 415-418.

[14] Hoppe H., DeRose T., Duehamp T., et al. Surface reconstruction from unorganized points. Computer Graphics, 1992, 26(2):71-78.