http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

OPEN ACCESS Information Technology Journal

ISSN 1812-5638
DOI: 10.3923/itj.2016.61.69

Research Article
A Parallel Computing Algorithm for Geometric Interpolation Using Uniform B-splines Within GPU

Hao Xu, L. Lu, W.K. Gu and Y.C. Zhou

Department of Computer Science and Engineering, South China University of Technology, Guangzhou, Guangdong, People's Republic of China

Abstract

Background: This study implemented an algorithm geometrically interpolating a given polygon using uniform B-splines with parallel computing within GPU by OpenCL. **Methodology:** Parallel matrix calculation and multi-partition methods were proposed to accelerate computing ability. **Results:** The results showed the implementation within GPU had greater advantage in computing than CPU. **Conclusion:** The GPU's implementation took less extra cost for improving the accuracy than CPU's. The GPU had great value on high precision applications with numerous interpolated points.

Key words: Geometric interpolation, parallel computing, GPGPU, OpenCL

Received: May 06, 2016 Accepted: May 23, 2016 Published: June 15, 2016

Citation: Hao Xu, L. Lu, W.K. Gu and Y.C. Zhou, 2016. A parallel computing algorithm for geometric interpolation using uniform B-splines within GPU. Inform. Technol. J., 15: 61-69.

Corresponding Author: Hao Xu, Department of Computer Science and Engineering, South China University of Technology, Guangzhou, Guangdong, People's Republic of China

Competing Interest: The authors have declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

Interpolating a parametric curve to scattered points had been widely investigated due to their extensive applications in a variety of fields such as economics, medicine, computer animation, manufacturing, CAGD and signal processing[1-6]. Though one may imposed different requirements on generated curves depending on applications[1,7], it was common that the interpolative curves should look pleasing and had little jerk, namely and they were as fair as possible.

Interpolation using parametric curves could state as follows: Given polygon (P_1, P_2, ..., P_m) and parametric curve:

$$C(t) = \sum_{i=1}^{m} P_i B_i(t)$$

where, B_i (t) and P_i were respectively base functions and unknown control points thus, needed to find $\{P_i\}$ and $t_1<t_2<...<t_m$ such that:

$$C(t_i) = P_i,\ i = 1, 2, \ldots, m$$

A straight forward method could directly specify $\{t_i\}$ and reduced the problem to the solution of a linear system on $\{P_i\}$. This method usually exhibited shortcomings such as overshooting and undulating. Hence, a great number of parameterization methods had been proposed to alleviate the artifacts[8-10]. Among these, arc-length parameterization and its variants were most popular in practice and efficiently alleviated the artifacts in most linear system. However, no solution could be viewed as the best in any case[11-13].

Hoschek[11] proposed an intrinsic parameterization to attack the curve-fitting problem. Parameter values computed implicitly by finding the nearest point from the data point to the fitting curve and then employed to fit a new curve. A nearest point was also called a foot-point. The approach iteratively refined the solution by repeating the process. Saux and Daniel[12] explored a new improvement of the method with higher efficiency[12] while Wang *et al.*[14] extended the approach with several new fitting errors. Unfortunately, these approaches degenerated to straightforward methods if applied to interpolating problems instead of fitting tasks. What was more important and computing overload weakened the process ability with time extending.

Recently, Maekawa *et al.*[15] exploited a geometric interpolation algorithm to go through vertices of a given mesh using subdivision surfaces[15]. The algorithm iteratively modified each control vertex of the mesh using a deviation from the foot-point to its corresponding data point. Maekawa *et al.*[15] discussed two types of foot-point, the nearest point on the subdivision surface to the data point and the limit position of the corresponding control point on the surface to the data point (Parametric distance deviation). As neither requiring solution of large linear systems nor yielding minimization of smooth energies, the approach was quite efficient in interpolating points. In addition, the approach could generally produce results with high quality using the first type of correction due to its essence of intrinsic parameterization. Chen *et al.*[16] extended the algorithm to Catumll Clark subdivision surfaces[16]. Gofuku *et al.*[17] further studied the interpolation of tangents and normal using the geometric algorithm[17]. Xiong *et al.*[18] investigated the convergence of geometric interpolation algorithm geometrically interpolating a given polygon and presenting a practical sufficient condition under which the algorithm was convergent, linear and time-saved, for quadratic and cubic B-splines, respectively[18]. In this implementations, within both CPU and GPU were under this sufficient condition.

When built an accurate curve model using B-splines and due with large quantity of data in which there would be many points to interpolate. It would take much time in the implementation of the geometric interpolation algorithm within CPU. As each step of geometric interpolation, using B-splines had to traverse all the interpolated points and the operations for each interpolated points were the same. Therefore, it was very compatible to implement geometric interpolation using B-splines within GPU and implement the algorithm by OplenCL. The process for OpenCL and implement the algorithm within GPU by OplenCL could design. To improve the performance even more, parallel matrix calculation, multi-partition method and elimination game algorithm used to accelerate the computing of the algorithm by parallel computing. Finally, in experiments, compared with traditional serial implementation of CPU showed the efficiency, GPU had great advantage in computing time and process ability. Where many interpolated points in GPU and there were less extra cost for improving the accuracy in finding the foot-point.

MATERIALS AND METHODS

Geometric interpolation: Let (P_1, P_2, ..., P_m) be a closed polygon and C (t) be the uniform cubic B-spline curve defined by (P_1, P_2, ..., P_m) with knot vector (-1, 0, 1, ..., m+2) where, the uniformity was not essential because the local parameterization polished up the curve shape under the

frame work of geometric interpolation. Therefore, uniform B-splines in this implementation used. Geometric interpolation iteratively modified the position of the control points such that finally the B-spline curve was determined by the new control points interpolate (P_1, P_2, ..., P_m).

Algorithm 1: Geometric interpolation using B-splines

Step 1: Initialized the control points P_i^0 with interpolated points, namely, set $(P_1^0, P_2^0, ..., P_m^0) = (P_1, P_2, ..., P_m)$

Step 2: Supposed that P_i^k was obtained. Constructed the parametric curve with the control points: P_i^k

$$C^k(t) = \sum_{i=1}^{m} P_i^k B_{i,n}(t)$$

where $B_{i,n}(t)$ were B-spline base functions with n-degree.

Step 3: Found the foot-point $\{P_i^k\}$ of $\{P_i^0\}$ on the curve

Step 4: Computed the deviation vectors:

$$e_i^k = P_i^k - P_i^0$$

Step 5: Checked error $e^k = \max_i \|e_i^k\|$ If $e^k < \epsilon$, it was finished. Else, modified the control points as follows:

$$P_i^{k+1} = P_i^k - e_i^k$$

and return to step 2.

In the case, the foot-point was a nearest point. It solved a non-linear optimization problem with a polynomial as the energy function. Subsequently, it needed to find zero points of a polynomial.

Parallel computing algorithm: As the computation for each interpolated points were the same and independent relatively and they could simply parallelize. However, it should be attention to control the data consistency. In algorithm 1, the computation of each step in all the processes should not start until all the computation before the step in all the processes had been finished.

For OpenCL, algorithm 1 with the process illustrated in Fig. 1. Sub-processes (1-5) could be computed parallel and there should be a data consistency control before each sub-process.

Matrix calculation: Because of using uniform B-splines, sub-process 2 was just a matrix multiplication for each segment of the parametric curve. In sub-processes 3 and 5, the computation of error vectors and the modification of control points were vector calculation. These computing and processing could implement independently for each element of matrices or vectors so that they could parallelize simply.

Since the independency, these computing and processing could parallelize combining with the first layer of parallelization. For OpenCL, it was just to add a dimension to the workspace for the first layer (For the interpolated points).

Multi-partition method: Let function $d_i(t)$ be the distance from the interpolated point P_i^0 to the point $C^k(t)$ that on the

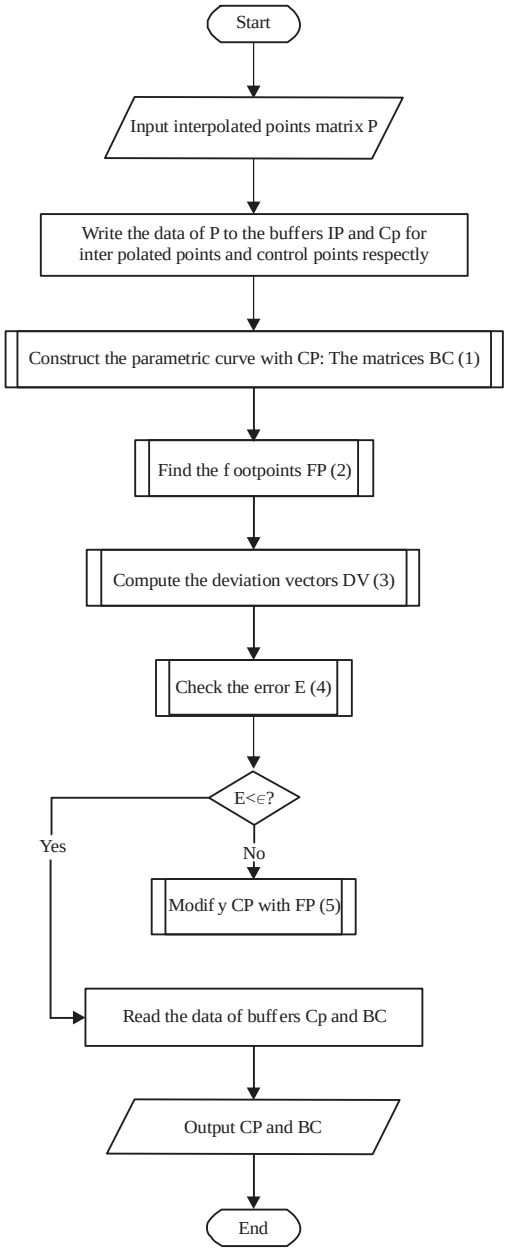

Fig. 1: Process of geometric interpolation using B-splines for OpenCL Furthermore, three kinds of computing and processing could accelerate through parallel methods. These were the second layer of parallelization

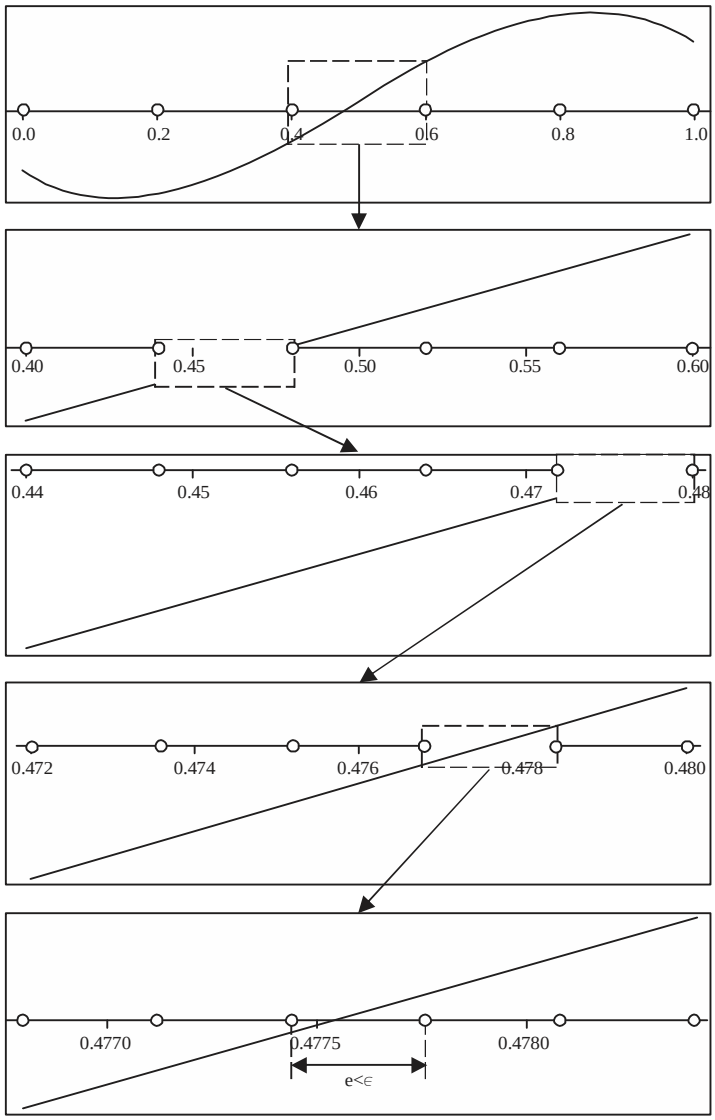

Fig. 2: Five partition method

curve. In sub-process (2), the foot-point of P_i^0, it was usually to calculate the roots of the differentiation $d_i'(t)$ and then calculate and compare the values of d_i (t) at these roots and the terminals of the domain to find the minimum one of them. For real-number function, there were many algorithms to calculate the roots of it. All these algorithms were to iteratively narrow down the interval which had a root to get the precise solution. Bisection method was the easiest one that it calculated the value of the midpoint in the interval and narrows down half of it.

Bisection method could not calculate even multiple roots, but the extreme points of d_i (t) were found which were not on the even multiple roots of $d_i'(t)$. It was not that all the extreme points were needed and just minimum points were needed. It was easy to prove that a function d_i (t) for n-degree B-spline had at most n+1 minimum points.

With parallel computing, the interval could partition to granular parts and calculated the values of the partition-points simultaneously in Fig. 2. However, the checking of the signs of the values had to be serial because it was not independence. Because the number of partition was not large and the comparing process was easier than the computation of the values and this method could be more efficient than bisection method with parallel computing though it had done more computing.

After the roots had found, calculation of the values of d_i (t) at these roots could parallelize. Nevertheless, the comparison of these values to find the minimum one also had

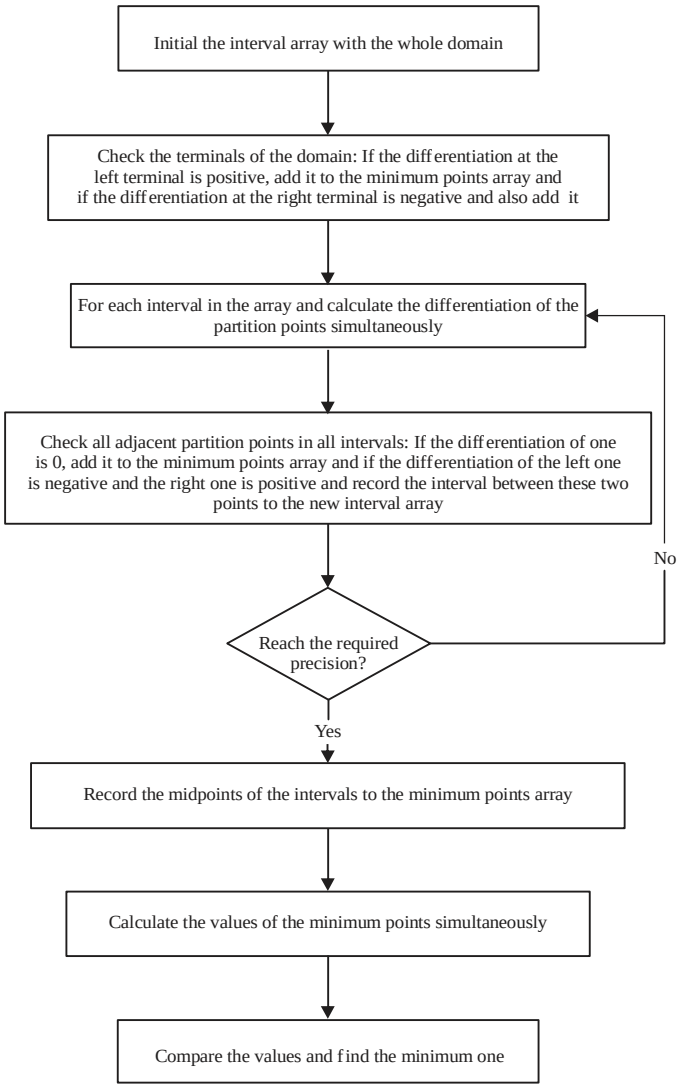

Fig. 3: Finding footpoints

to be serial and was easy. There were at most 2 and 3 minimum points (Including the terminals of the domain and the numbers were 3 and 4) for using quadratic and cubic B-splines, respectively.

For OpenCL, an enough large array should be created to record the intervals that a root in it and an integer to record the number of intervals. An array should also be created to record the minimum points that should be calculated the deviation values and an integer to record the number of them. For instance, an array should be created with 6 numbers (2 numbers to record an interval) to record the intervals and an array with 4 numbers to record the minimum points for using cubic B-splines. Sub-process 2 implemented and illustrated in Fig. 3.

Elimination game algorithm: In sub-process 4, the computation of norms $\|e_i^k\|$ of each deviation vector in DV could parallelize, but the comparing processing could not. Although the comparing processing was not hard, it took much time when there were many interpolated points. A method like an elimination game used to search the maximum with parallel computing in Fig. 4. Each round of the game could parallelize. Nevertheless, the processing should ensure the data consistency.

There should be a synchronization control before each round of the game. So, it also took some computing time on these synchronization controls. The number of values increased to compare in one game to take the balance with the cost of synchronization controls (Fig. 5). For different

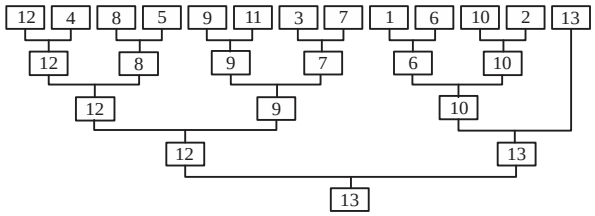

Fig. 4: Elimination game algorithm

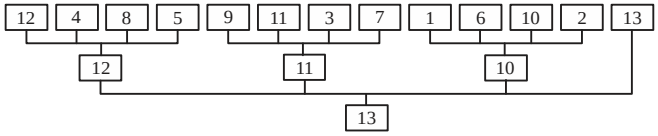

Fig. 5: Elimination game algorithm (4 values in one match) Although it was like a sorting algorithm, the data need not to sort and just need to get the maximum, namely. In the implementation, the maximum value in each match would be record in the space of the front value. Algorithm 2 in C-code is given

platforms and devices and the number of values should adjust in one match to get the best performance. This method was more efficient in the case with large number of interpolated points.

Algorithm 2: Elimination game algorithm

```
//E is the array of errors.
//n is the total number of values.
//m is the number of values in one match.
//There are (n/m+1) compute units.

for (i = 1; i<n; I* = m)
  {
      if ( compute_unit_id% i == 0)
      {
            o = i*m*compute_unit_id;
            for (j = o+i; j<n && j<o+i *m; j+ = i)
                  E [o] = E [o]>E [o+j] ? E [o]:E [o+j];
      }
      Synchronization();
}
```

All the computing of the algorithm could put into GPU, but there is a limit of compute units by the devices. In addition, the synchronization control in GPU would take much cost of time. Therefore, the outer loop control put in CPU and implemented the synchronization by OpenCL events (Command queue in order execute mode).

These three methods had not reduced the amount of calculation and multi-partition method in searching foot-point even increased it. Nevertheless, within parallel implementation, these three methods could decrease the cost of time.

RESULTS

These experiments focused on the computing speed contrasting between implementation within GPU and traditional implementation within CPU (Table 1). Repetitious computing could be in each case and record the average computing time.

OpenCL platform used for GPU programming and perform these experiments on a computer as follow:

DISCUSSION

To observe the efficiency, some sets of random points for universality generated. All these points had 3-dimensions and were placed in cube [0, 1]×[0, 1]×[0, 1]. Double precision number data type used to set the precision. Each set of points and its experimental results were as following:

- Computing time for different numbers of values was in one match for elimination game algorithm. A data with 50000 values used to test the algorithm within GPU. The results listed in Table 2 and illustrated in Fig. 6. It was obvious that setting the number of values in one match about 30 could take the balance with the cost of synchronization controls and perform most efficiently within GPU on the environment of this experiment
- Computing time for different numbers of partitions was in multi-partition method. A data with 10000 points was used and set the precision for the multi-partition method to be . The number of values in one match for elimination

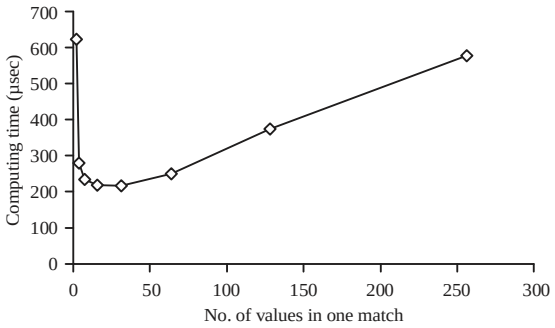

Fig. 6: Computing time for different iterative times

Table 1: Experiment platform

CPU	Intel Core i7 860 (2.8 GHz)
Memory	8G DDR3 1333 MHz
GPU	NVIDIA GeForce GTX 580 (772 MHz)
GPU Memory	1536M GDDR5
OS	Windows 7 Ultimate 64bit
OpenCL Platform	NVIDIA CUDA 4.2

Table 2: Computing time for different number of values in one match for elimination game algorithm

No. of values in one match	2	4	8	16	32	64	128	256
Computing time (μsec)	642	281	234	219	218	250	375	577

Table 3: Experiment platform

No. of partitions	2	3	7	15	31	63
CPU computing time (msec)	7457	7004	6739	9173	12948	35038
GPU computing time (msec)	18658	12777	8580	7815	12496	18642
No. of partitions	127	255	511	1023	2047	
CPU computing time (msec)	67548	108592	210117	603065	1186830	
GPU computing time (msec)	128455	18642	54772	85676	169556	

game algorithm was 50. The results listed in Table 3 and illustrated in Fig. 7. It shown that 15-partition was the best choice to perform most efficiently within GPU on the environment of this experiment. Within CPU, it performed best with 7-partition in this experiment, not with bisection (2-partition) because of the cost of recursive calling. It meant that similar method could be also used to optimize the performance of some CPU programs in which there were recursive calling. Although, the results of big numbers were useless, the capability of GPU for parallel computing contrasting to CPU could be seen

- Computing time for different precisions needed the multi-partition method. A data with 10000 points used and took bisection method for CPU and 15-partition method for GPU. The number of values in one match for elimination game algorithm was 50. The results listed in Table 4 and illustrated in Fig. 8. Here the whole slopes of the curves in Fig. 8 could be focused which represent the extra cost of accuracy improvement within each implementation. It was obvious that the implementation of GPU here took less extra cost to improve the accuracy with 15-partition method
- Computing time was different with numbers of points. 7-partition method for CPU was used, 15-partition method for GPU and set the precision for the multi-partition method to be 10^{-9}. The number of values in one match for elimination game algorithm was 50. The results listed in Table 5 and illustrated in Fig. 9. The experimental results demonstrated that the implementation of GPU had great advantage on computing time in the cases with large number of points because of the great data parallel computing capability of GPU. This was the main reason that the GPU should be choose. In the cases with small number of points, CPU perform was more efficiently than GPU due to its higher frequency (About 4 times of GPU's) but in the cases of large number of points, GPU perform was better since the parallel computing and that GPU was fit to do these parallel computing with large quantity of data

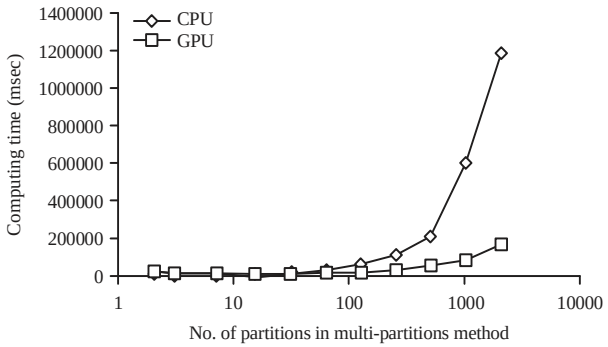

Fig. 7: Computing time for different numbers of partitions in multi-partition method, the abscissa is using logarithmic coordinate

Table 4: Experiment platform

Precision for multi-partition method	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
CPU computing time (msec)	4072	4868	5492	9173	10764	11997	13276
GPU computing time (msec)	4665	5335	6068	6771	6708	7503	8175

Table 5: Experiment platform

No. of values in one match	10	100	1000	10000	20000
CPU computing time (msec)	10	94	1279	6099	18782
GPU computing time (msec)	78	119	1778	7847	14648
No. of values in one match	30000	40000	50000	60000	
CPU computing time (msec)	29282	48001	71714	92876	
GPU computing time (msec)	21278	28064	34913	41059	

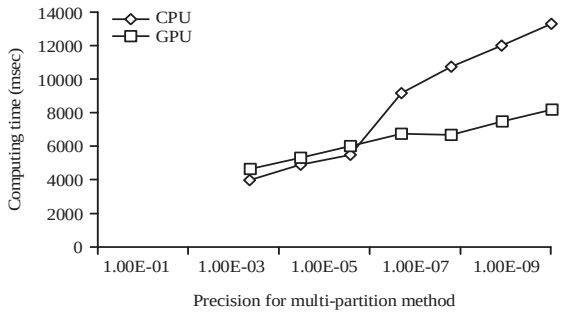

Fig. 8: Computing time for different number of interpolated points

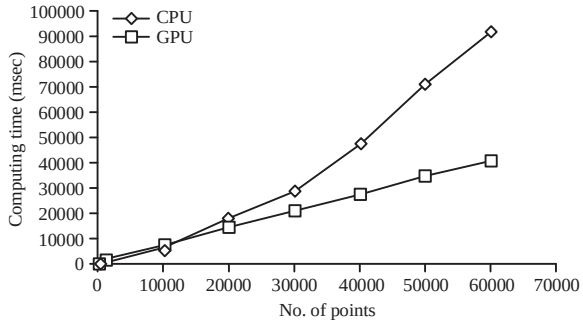

Fig. 9: Computing time for different number of points

- Compared with other GPU optimizations, LFR network datasets were chosen for the artificial network datasets which was proposed by Lancichinetti and Fortunato[19]. It could strengthen the results that GPU perform was better in the parallel computing. The scale of the network ranges from about 100-50000. Now-a-days, it had been widely used as the standard datasets. As for the real network, some common large network data were adopted, including Email, Facebook, the data of Scientists Websites--Netscience and Slovenia scientists, national grid data and so on. However, these network datasets were all for social network, it had not been used in uniform B-splines optimizations

CONCLUSION

Geometric interpolation, iterative algorithm to construct control polygons and meshes. However, they have not run on GPU devices. This study applied this algorithm using uniform B-splines with parallel computing within GPU. To improve the performance even more, parallel matrix calculation, multi-partition method and elimination game algorithm were proposed to accelerate the computing of the algorithm by parallel computing.

According to the experimental results, the parallel computing implementation of geometric interpolation using uniform B-splines within GPU had great advantage of computing time than CPU in the cases with large quantity of data. In addition, with parallel computing and multi-partition method, GPU's implementation took less extra cost for improving the accuracy than CPU's. It had great value on high precision applications with numerous interpolated points or large linear systems.

Non-uniform B-spline would be considered in the future study. Constructing curve would not be just a matrix multiplication using non-uniform B-spline. Another interesting work was geometric interpolation for parametric surfaces and subdivision surfaces.

ACKNOWLEDGMENT

This study was supported by Guangdong Production, Education and Research Project (2015B010107001), Guangzhou Production, Education and Research Project (201508010034).

REFERENCES

1. Hagan, P.S. and G. West, 2006. Interpolation methods for curve construction. Applied Math. Finance, 13: 89-129.
2. Jacob, M., T. Blu and M. Unser, 2004. Efficient energies and algorithms for parametric snakes. IEEE Trans. Image Process., 13: 1231-1244.
3. Ramamoorthi, R. and A.H. Barr, 1997. Fast construction of accurate quaternion splines. Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, August 3-8, 1997, Los Angeles, CA., pp: 287-292.
4. Nam, S.H. and M.Y. Yang, 2004. A study on a generalized parametric interpolator with real-time jerk-limited acceleration. Comput. Aided Design, 36: 27-36.
5. Wang, F.C. and D.C.H. Yang, 1993. Nearly arc-length parameterized quintic-spline interpolation for precision machining. Comput. Aided Design, 25: 281-288.
6. Levin, A., 1999. Interpolating nets of curves by smooth subdivision surfaces. Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, August 8-13, 1999, Los Angeles, CA., pp: 57-64.
7. Piegl, L.A., K. Rajab and V. Smarodzinana, 2008. Curve interpolation with directional constraints for engineering design. Eng. Comput., 24: 79-85.
8. Floater, M.S. and T. Surazhsky, 2006. Parameterization for curve interpolation. Stud. Comput. Math., 12: 39-54.
9. Papaioannou, S.G. and M.M. Patrikoussakis, 2011. Curve interpolation based on the canonical arc length parametrization. Comput. Aided Design, 43: 21-30.

10. Kouibia, A., M. Pasadas and M.L. Rodriguez, 2011. Optimization of parameters for curve interpolation by cubic splines. J. Comput. Applied Math., 235: 4187-4198.
11. Hoschek, J., 1988. Intrinsic parametrization for approximation. Comput. Aided Geometric Design, 5: 27-31.
12. Saux, E. and M. Daniel, 2003. An improved Hoschek intrinsic parametrization. Comput. Aided Geometric Design, 20: 513-521.
13. Pottmann, H. and S. Leopoldseder, 2003. A concept for parametric surface fitting which avoids the parametrization problem. Comput. Aided Geometric Design, 20: 343-362.
14. Wang, W., H. Pottmann and Y. Liu, 2006. Fitting B-spline curves to point clouds by curvature-based squared distance minimization. ACM Trans. Graphics, 25: 214-238.
15. Maekawa, T., Y. Matsumoto and K. Namiki, 2007. Interpolation by geometric algorithm. Comput. Aided Design, 39: 313-323.
16. Chen, Z., X. Luo, L. Tan, B. Ye and J. Chen, 2008. Progressive interpolation based on catmull-clark subdivision surfaces. Comput. Graphics Forum, 27: 1823-1827.
17. Gofuku, S.I., S. Tamura and T. Maekawa, 2009. Point-tangent/point-normal B-spline curve interpolation by geometric algorithms. Comput. Aided Design, 41: 412-422.
18. Xiong, Y., G. Li and A. Mao, 2011. Convergence of geometric interpolation using uniform B-splines. Proceedings of the 12th International Conference on Computer-Aided Design and Computer Graphics, September 15-17, 2011, Jinan, pp: 229-234.
19. Lancichinetti, A. and S. Fortunato, 2009. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. Physical Rev. E, Vol. 80. 10.1103/PhysRevE.80.016118.