Parallelized Force-Directed Edge Bundling on the GPU

Delu Zhu
Computer Network Information Center, Chinese Academy of Sciences
Graduate University of Chinese Academy of Sciences
Beijing, 100190, China
jasmine@cnic.cn

Kaichao Wu, Danhuai Guo
Computer Network Information Center, Chinese Academy of Sciences
Beijing, 100190, China
kaichao@cnic.cn,
guodanhuai@cnic.cn

Yuanmin Chen
China Internet Network Information Center
DNSLAB
Beijing, 100190, China
chenyuanmin@cnnic.cn

***Abstract*—How to draw large scale spatial interaction data clearly and quickly is a challenge in high performance data visualization research and application field. Force-Directed Edge Bundling (FDEB) helps display graph clearly with significant clutter reduction, but with high time complexity. This paper presents a parallelized FDEB on the GPU (GPU-FDEB), which reforms FDEB and achieves a balanced partitioning of data and calculation to suit computation on the GPU. GPU-FDEB addresses the problem of high time complexity and accelerates FDEB by an order of magnitude.**

Keywords-GPU; parallel; FDEB; visualization; CUDA

I. INTRODUCTION

Graphs are widely used to model relationships among data such as connections between people, traffic between locations and migration between counties. An intuitive way to visualize graphs is using node-link diagram, however, when graphs comprised of a large number of nodes and edges are visualized as node-link diagrams, visual clutter quickly becomes a problem (see Fig. 8a) [1]. Many methods have been proposed to reduce visual complexity [6-10]. These methods can be classified into two major categories: adjust node positions and improve edge layout [2].

Hierarchical Edge Bundling (HEB), Geometry-Based Edge Bundling (GBEB) and Force-directed Edge Bundling (FDEB) proposed in [1, 2, 3] reduce visual compatibility by adjusting edges (merging similar edges into bundles). HEB is limited to visualize graphs with a hierarchical structure [3]. GBEB is suitable for general graphs, but need to generate a control mesh to guide the edge clustering process [2]. While FDEB is suitable for general graphs and don't require a hierarchy or a control mesh [1]. FDEB is a self-organizing edge-bundling approach in which edges are modeled as flexible springs that can attract each other and be grouped into bundles [1].

FDEB can reduce visual complexity with a significant clutter reduction, but the computational complexity of FDEB is O(N·M^2·K), with N = iterations, M = edges, and K = subdivision points per edge [1]. When the number of edges comes to 15000, the runtime of FDEB will exceed 6 minutes. The high computational cost of large graphs is the major drawback of FDEB. We may consider parallelize FDEB using multi-core platforms or clusters to achieve real-time response. However, it's not only difficult to design multi-core or do deployment in a clustered environment but also costly [17]. While GPU has hundreds of cores and low cost. Furthermore, CUDA (Compute Unified Device Architecture) makes GPGPU (General-Purpose computing on GPU) easier and more available to end users [4, 11, 12, 16].

We present a parallelized FDEB on the GPU (GPU-FDEB) according to some criterion such as maximize the independence between threads and maximize the computational intensity to achieve a maximum speedup.

The remainder of this paper is organized as follows. In Section 2 we give an overview of FDEB. Section 3 presents the algorithm of parallelized FDEB. Section 4 describes the implementation of our algorithm on the GPU and the accelerate technique in detail. In Section 5 we present experimental results and analysis. Finally, Section 6 presents conclusions and directions for future work.

II. FORCE-DIRECTED EDGE BUNDLING

Take population migration graph for example (see Fig. 8a), in which nodes depict migration locations and edges depict population migrated between locations. In this node-link diagram comprised of a large number of edges, visual clutter is such a serious problem that we can't detect any visible high-level migration patterns. FDEB can achieve significant clutter reduction in visualizing large graphs (see Fig. 8b).

In FDEB, edges are subdivided into segments. The positions of end-points of all edges remain fixed and the position of each subdivision point is changing by the combined force of spring force and electrostatic force while bundling [1].

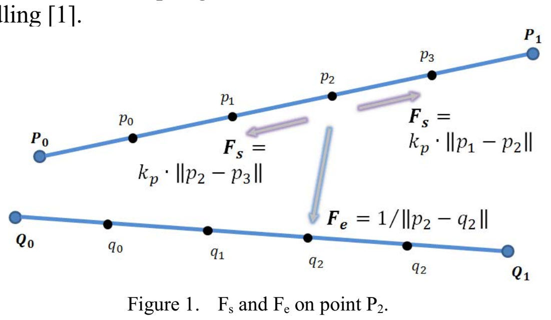

Figure 1. F_s and F_e on point P_2.

Take Fig. 1 for example, there are two spring forces (F_s for short) exerted on subdivision point p_2 by p_1, p_3 and one electrostatic force (F_e for short) exerted on p_2 by q_2. F_s is used between each pair of consecutive subdivision points on the same edge, and F_e is used between each pair of corresponding subdivision points of a pair of interacting edges [1].

In the process of bundling, each subdivision point of edges makes a step in the direction of the combined force exerted on it during each iterative step, which will iterate many times until the edges are bundled sufficiently [1].

Furthermore, to control the amount of interaction between edges, FDEB introduce the concept of edge compatibility [1]. A pair of edges P and Q is considered as interactive if edge compatibility $C_e(P, Q)$ between P and Q is above a certain threshold, which reduces the amount of bundling desirably and the computation largely [1].

In [1] the combined force F_{pi} exerted on subdivision point pi on edge P is defined as:

$$F_{p_i} = k_p \cdot (\|p_{i-1} - p_i\| + \|p_i - p_{i+1}\|) + \sum_{Q \in E} \frac{C_e(P,Q)}{\|p_i - q_i\|},$$

With
k_p: spring constant for each segment of edge P,
E: set of all interacting edges except edge P.

III. GPU-FDEB ALGORITHM

The crucial way to improve the efficiency of GPU-FDEB is to properly distribute tasks between CPU and GPU [4, 11, 12, 15]. FDEB uses an iterative refinement scheme to calculate the bundling [1]. The calculation of spring force, electrostatic force and the moving of subdivision points are coupled together at each iteration step, which is not suitable for GPU computing, so we reform the calculation of each iteration step into 3 independent steps: calculate spring force, calculate electrostatic force, and update the positions of subdivision points.

Fig. 2 displays the flow chart of GPU-FDEB, in which the blue blocks (1, 3.1, 4) are executed on the CPU, while the green blocks (2, 3.2, 3.3.1, 3.3.2, 3.3.3) which are of high computational cost are executed on the GPU.

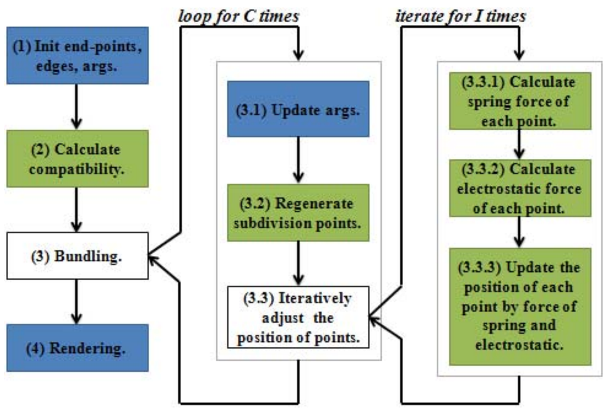
loop for C times
iterate for I times
(1) Init end-points, edges, args.
(2) Calculate compatibility.
(3) Bundling.
(4) Rendering.
(3.1) Update args.
(3.2) Regenerate subdivision points.
(3.3) Iteratively adjust the position of points.
(3.3.1) Calculate spring force of each point.
(3.3.2) Calculate electrostatic force of each point.
(3.3.3) Update the position of each point by force of spring and electrostatic.

Figure 2. The algorithm flow of GPU-FDEB.

The algorithm is described in details as follows:

(1) Initialize:

(1.1) Input nodes and edges.

(1.2) Initialize the parameters such as the number of iteration steps I, the number of subdivision points P, the step size S.

(1.3) Allocate device memory, transfer edges from CPU to GPU.

(2) Calculate edge compatibility for each combination of edges on the GPU: Thread (i, j) is responsible to calculate edge compatibility between edge i and edge j.

(3) Bundling:

```
For cycle: =1 to C do
Begin
(3.1) Update parameters: P is increased by a fixed rate, I and S are decreased by a fixed rate.
(3.2) Regenerate subdivision points:
    (3.2.1) Thread (i, j) is responsible to calculate the length of segment j on edge i.
    (3.2.2) Thread (i, 0) is responsible to generate P subdivision points on edge i.
(3.3) Iteratively adjust the positions of subdivision points:
    For iterator: =1 to I do
    Begin
    (3.3.1) Calculate spring force: Thread (i, j) is responsible to calculate two neighboring F_s exerted on subdivision point j on edge i.
    (3.3.2) Calculate electrostatic force: Thread (i, j) is responsible to calculate the sum of F_e exerted on subdivision point j on edge i by corresponding subdivision points on edges interacted with edge i.
    (3.3.3) Update the position: Thread (i, j) is responsible to move the subdivision point j on edge i to a distance of S in the direction of the combined force exerted on it.
    End.
End.
```

(4) Rendering.

IV. GPU IMPLEMENTATION

A. Data storage

The layout of data in memory has great influences on the efficiency of GPU-FDEB [12, 15]. In GPU-FDEB, the graph is represented as a matrix where all the points on each edge are placed in each row (see Fig. 3), and the matrix is stored in FORTRAN order in memory, i.e., all the 1st points of each edge are stored in sequence, so are the 2nd points and the 3rd points, etc. Furthermore, the coordinate X and Y of all the points are separately stored in two matrixes.

We choose this data layout for two reasons. First, the number of subdivision points P is not fixed but increased with increasing cycles. If we store the matrix by row, we have to modify each row every time we add new subdivision points. Second, since the major calculation of GPU-FDEB is processed column by column, there is no dependency between columns, thus storing points by column can maximize memory access locality. e.g., we can load the whole column into cache and manipulate the elements there instead of accessing the elements one by one from memory.

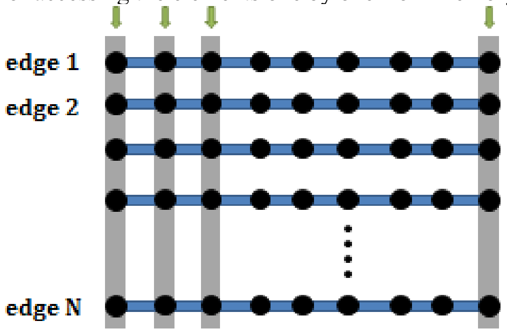

Figure 3. The points are stored by column.

Besides, to hide memory access latency, we expand the column dimension into multiples of 16, thus the starting address of each column is times of 64 (16×4) bytes, which meets the requirement of memory coalescing, i.e., 16 simultaneous memory accesses can be coalesced into a single memory transaction.

B. *Calculate edge compatibility*

This kernel is responsible to calculate compatibility between every two edges. The input is the end-points and lengths of N edges, and the output is an N×N compatibility matrix (N is the number of edges after expanded, times of 16). Considering that compatibility matrix is symmetric, we only need to calculate the lower triangular matrix.

Since row dimension of compatibility matrix is N (times of 16) and we store compatibility matrix by row, thus it is memory coalesced to write the lower triangular matrix back into global memory after the calculation of edge compatibility.

C. *Regenerate subdivision points*

This kernel is responsible to regenerate subdivision points for each edge. The input is the data matrix comprised of end-points and subdivision points of N edges, and the output is the regenerated subdivision points of N edges. The layout of data matrix in Fig. 3 is specified as Fig. 4 shows. The 2nd end- points of N edges are stored in the 1st column, the 1st end-points of N edges are stored in the 2nd column and the regenerated subdivision points are stored in the rest columns.

We do this for two reasons. First, the number of subdivision points will increase with increasing cycles, it's reasonable to store the constant end-points in the first two columns while leave the variable subdivision points behind. Second, it maximizes memory access locality to store the 1st end-points close to the 1st subdivision points.

From step (3.2) of GPU-FDEB algorithm, to calculate the positions of regenerated subdivision points on each edge, we need to calculate and sum the length of each segment on each edge (the red segments showed in Fig. 4) to get the length of each edge, then get the positions of the points which divide edges equally. It can be seen that calculating the length of each segment is independent with each other and so is regenerating subdivision points on each edge. A large parallelism is achieved in this kernel.

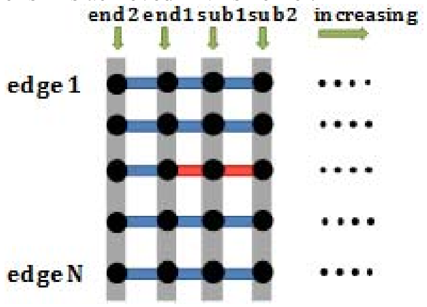

Figure 4. Generate subdivision points.

D. *Calculate electrostatic force*

This kernel is responsible to calculate the total electrostatic force exerted on each subdivision point, which is the most time-consuming part in FDEB. The input is the data matrix comprised of end-points and subdivision points, the weights of N edges and the compatibility matrix, while the output is a matrix of the total electrostatic force exerted on each subdivision point. Given the concept of electrostatic force defined in FDEB (see Section 2), the total electrostatic force exerted on subdivision point j in Fig. 5 is the sum of the electrostatic force exerted on j by each point in the same column as j.

Since there is no dependency among columns in this kernel, all the columns can be processed in parallel. To calculate the total electrostatic force exerted on each subdivision point in each column, we need to calculate the electrostatic force F_e for every two points in this column, i.e. we need to calculate an N×N F_e matrix. The F_e is interactive which means that the F_e matrix is symmetric, so we only need to calculate the lower triangular matrix (the green part in Fig. 6) which reduces the calculation largely.

To obtain the total electrostatic force exerted on each subdivision point, we need to calculate the sum of each row in the F_e matrix. However, we only calculated the lower triangular matrix, it's impossible to sum each row directly. Considering that the F_e matrix is symmetric, the uncalculated upper triangular matrix can be obtained from the calculated lower triangular matrix. As Fig. 7 shows, the sum of blue part in the left matrix equals the blue part in the right matrix.

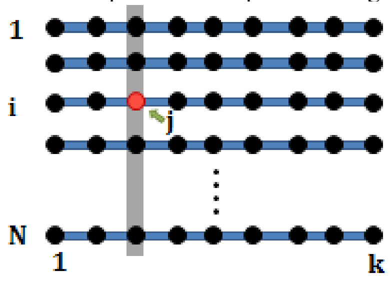

Figure 5. Calculate the electrostatic force.

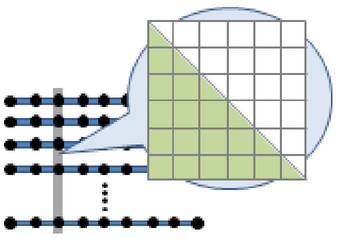

Figure 6. Each column corresponds to a F_e matrix.

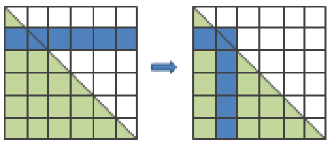

Figure 7. Map the uncalculated part to calculated.

V. EXPERIMENTAL RESULTS

We have carried on detailed experiments on our GPU-FDEB algorithm. The experiment platform is showed in Table 1.

TABLE I. EXPERIMENT PLATFORM

	Model	Cores	Frequency	Cache	Memory
CPU	Intel Xeon X5650	6×2	2.67GHz	12M	10G
GPU	Nvidia Quadro 5000	352×1	1.03GHz	655K	2.56G
Runtime	RHEL 6, CUDA Driver Version 4.0				

The experimental data are obtained from population migration between counties of US (3219 node, 375374 edges). The maximum number of edges can be processed on our GPU is between 15000 and 16000, which is limited by the size of GPU memory. So we select a subgraph comprised of 15000 edges and 372 nodes as our test graph and process it with FDEB and GPU-FDEB respectively. Fig. 8a shows the original graph. Fig. 8b and Fig. 8c shows the bundled graph processed by FDEB and GPU-FDEB respectively. Fig. 8a and Fig. 8b show that FDEB has reduced visual clutter largely and reveal high-level migration patterns. Fig. 8b and Fig. 8c show that GPU-FDEB gets almost the same bundled graph as FDEB.

To measure the efficiency of GPU-FDEB, we record the time taken by FDEB and GPU-FDEB to calculate the bundled graph respectively as the number of edges changes. All the time accepted in Fig. 9 is an average of ten successive times.

In Fig. 9, we can see that the time consumed by FDEB increases rapidly as the number of edges increases, while slowly for GPU-FDEB. In terms of time-consumption, the advantage of GPU-FDEB is more obvious with the number of edges increasing. When it comes to 4000 edges, the time consumed by FDEB is more than 19 seconds, while less than 3 seconds by GPU-FDEB. When it comes to 15000 edges, FDEB takes more than 6 minutes while GPU-FDEB only need 36 seconds.

Fig. 10 shows the variation of speedup (the ratio of time consumed in FDEB to GPU-FDEB) as the number of edges increases. The speedup is increasing gradually as the number of edges increases until it comes to 2000 edges, followed by a gradual decrease until it comes to 5000 edges, and then has an ascent trend in the end. The drop of speedup around 5000 edges indicates that there's a decrease in GPU performance.

Overall, our GPU-FDEB algorithm is reliable and effective.

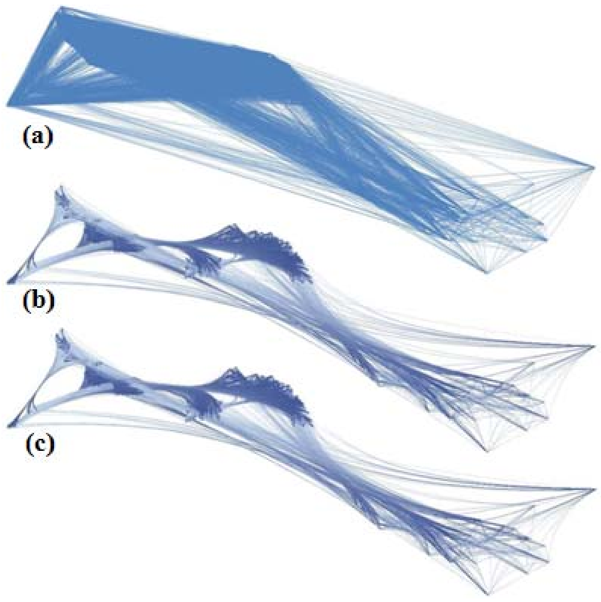

Figure 8. Population migration graph (a)origin(b)FDEB(c)GPU-FDEB.

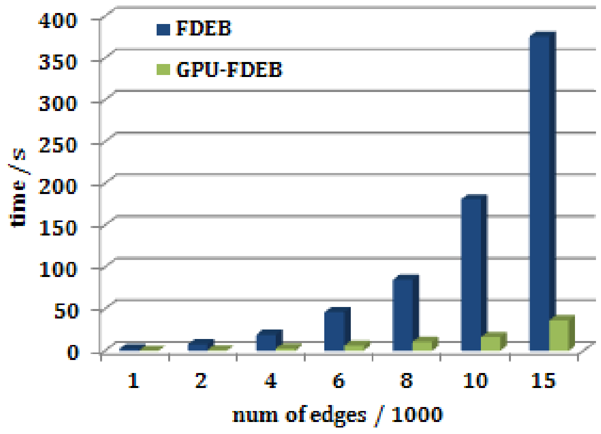

Figure 9. Time consumption.

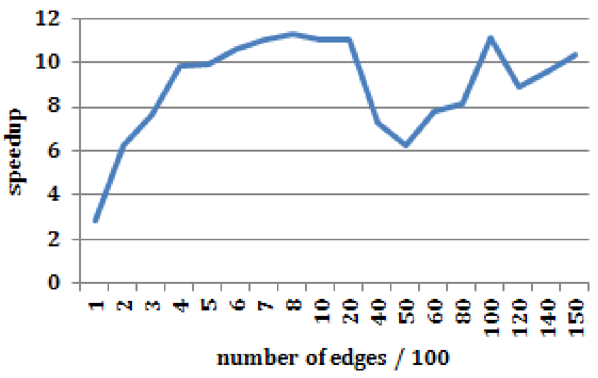

Figure 10. Speedup over FDEB.

VI. CONCLUSION AND FUTERWORK

This paper has presented a parallelized FDEB implemented on the GPU. Furthermore, we have optimized the layout of data in memory to maximize memory access locality. Most memory accesses have been coalesced to reduce memory access latency. Compared with FDEB implemented in C, GPU-FDEB achieves a speedup of 11 at most on the platform showed in table 1.

The GPU-FDEB implemented in this paper only support single GPU, thus the quantity of data can be processed is limited by hardware resource (e.g., the number of SM and the size of GPU memory). A straightforward way to address this is to use multi-GPU system or GPU cluster, which we would implement in our next work. Furthermore, it is possible to split edges of general graph into blocks according to edge compatibility, which could be used in multi-GPU system or GPU cluster.

ACKNOWLEDGMENT

This work is supported by National Development and Reform Commission of China in project "Scientific Research Information Infrastructure Construction and Typical Demonstrations Based on CNGI", Chinese Academy of Sciences #CNIC_ZR_201201, and DNSLAB of China Internet Network Information Center #DNSLAB-2012-D-U.

REFERENCES

[1] D. Holten and J. J. van Wijk, "Force-directed edge bundling for graph visualization," Computer Graphics Forum, vol. 28, pp. 983-990, 2009.

[2] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li, "Geometry-based edge clustering for graph visualization," IEEE Transactions on Visualization and Computer Graphics, vol. 14, pp. 1277-1284, 2008.

[3] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," IEEE Transactions on Visualization and Computer Graphics, vol. 12, pp. 741–748, 2006.

[4] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010.

[5] A. Grama, A. Gupta, G. Karypis, and V. Kumar, Introduction to Parallel Computing, Addison Wesley, 2nd ed, 2003.

[6] G. Ellis and A. Dix, "A taxonomy of clutter reduction for information visualisation," IEEE Transactions on Visualization and Computer Graphics, vol. 13, pp. 1216-1223, 2007.

[7] F. van Ham and J. J. van Wijk, "Interactive visualization of small world graphs," IEEE Symposium on Information Visualization, pp. 199-206, 2004.

[8] A. Noack, "An energy model for visual graph clustering. In Proceed. of Symposium on Graph Drawingg," pp. 425-436, 2003.

[9] N. Wong, M. Carpendale, and S. Greenberg, "Edgelens: An interactive method for managing edge congestion in graphs," IEEE Symposium on Information Visualization, pp. 51-58, 2003.

[10] H. Qu, H. Zhou, and Y. Wu, "Controllable and progressive edge clustering for large networks," In Proceed. of Symposium on Graph Drawing, pp. 399-404, 2006.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," Proc. Proceedings of IEEE, 2008,pp. 879-899, doi:10.1109/JPROC.2008.917757.

[12] NVIDIA, nVidia CUDA Programming Guide 3.0, NVidia, 2010.

[13] NVIDIA, The CUDA Compiler Driver NVCC, NVidia, 2008.

[14] NVIDIA, CUDA-GDB: The NVIDIA CUDA Debugger User Manual 2.1, 2008.

[15] M. Garland, "Parallel computing with CUDA," Proc. IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010,pp. 1, doi:10.1109/IPDRS.2010.5470378.

[16] J. Nickolls and W. J. Dally, "The GPU Computing Era," Proc. Micro, IEEE, vol. 30, 2010,pp. 56-69, doi:10.1109/MM.2010.41.

[17] Quinn and J. Michael, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Science, 2004.

[18] Y. Frishman and A. Tal, "Multi-level graph layout on the GPU," IEEE Transactions on Visualization and Computer Graphics, vol. 13, pp. 1310-1319, 2007.